16th Symposium on Experimental Algorithms

SEA 2017, June 21–23, 2017, London, United Kingdom

Edited by

Costas S. Iliopoulos
Solon P. Pissis
Simon J. Puglisi
Rajeev Raman

Editors

Costas S. Iliopoulos
King's College London
London, UK
csi@kcl.ac.uk

Solon P. Pissis
King's College London
London, UK
solon.pissis@kcl.ac.uk

Simon J. Puglisi
University of Helsinki
Helsinki, Finland
puglisi@cs.helsinki.fi

Rajeev Raman
University of Leicester
Leicester, UK
r.raman@leicester.ac.uk

ACM Classification 1998
F.2 Analysis of Algorithms and Problem Complexity, I.1.2 Algorithms

ISBN 978-3-95977-036-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-036-1.

Publication date
August, 2017

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): http://creativecommons.org/licenses/by/3.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.SEA.2017.0

ISBN 978-3-95977-036-1 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Luca Aceto (Reykjavik University)
- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Preface
Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman 0:ix

Invited Papers

Designing Energy-Efficient Heat Recovery Networks using Mixed-Integer Nonlinear Optimisation
Radu Baltean-Lugojan, Christodoulos A. Floudas, Ruth Misener, and Miten Mistry 1:1–1:1

Dictionaries Revisited
Martin Farach-Colton 2:1–2:1

Engineering Streaming Algorithms
Graham Cormode 3:1–3:1

Regular Papers

Better Process Mapping and Sparse Quadratic Assignment
Christian Schulz and Jesper Larsson Träff 4:1–4:15

The Isomap Algorithm in Distance Geometry
Leo Liberti and Claudia D'Ambrosio 5:1–5:13

Distributed Domain Propagation
Robert Lion Gottwald, Stephen J. Maher, and Yuji Shinano 6:1–6:11

Efficient Algorithms for k-Regret Minimizing Sets
Pankaj K. Agarwal, Nirman Kumar, Stavros Sintos, and Subhash Suri 7:1–7:23

Engineering an Approximation Scheme for Traveling Salesman in Planar Graphs
Amariah Becker, Eli Fox-Epstein, Philip N. Klein, and David Meierfrankenfeld .. 8:1–8:17

Approximating the Smallest 2-Vertex-Connected Spanning Subgraph via Low-High Orders
Loukas Georgiadis, Giuseppe F. Italiano, and Aikaterini Karanasiou 9:1–9:16

Extending Search Phases in the Micali-Vazirani Algorithm
Michael Huang and Clifford Stein 10:1–10:19

A Framework of Dynamic Data Structures for String Processing
Nicola Prezza 11:1–11:15

Practical Range Minimum Queries Revisited
Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit 12:1–12:16

Compression with the tudocomp Framework
Patrick Dinklage, Johannes Fischer, Dominik Köppl, Marvin Löbel, and Kunihiko Sadakane 13:1–13:22

Algorithm Engineering for All-Pairs Suffix-Prefix Matching
Jihyuk Lim and Kunsoo Park 14:1–14:12

16th Symposium on Experimental Algorithms.
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman

LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The Quantile Index – Succinct Self-Index for Top-k Document Retrieval
Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit 15:1–15:14

Online Construction of Wavelet Trees
Paulo G. S. da Fonseca and Israel B. F. da Silva 16:1–16:14

Engineering External Memory LCP Array Construction: Parallel, In-Place and Large Alphabet
Juha Kärkkäinen and Dominik Kempa 17:1–17:14

Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees
Stefan Funke, Sören Laue, and Sabine Storandt 18:1–18:13

Consumption Profiles in Route Planning for Electric Vehicles: Theory and Applications
Moritz Baum, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf 19:1–19:18

Efficient Traffic Assignment for Public Transit Networks
Lars Briem, Sebastian Buck, Holger Ebhart, Nicolai Mallig, Ben Strasser, Peter Vortisch, Dorothea Wagner, and Tobias Zündorf 20:1–20:14

Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure
Tobias Heuer and Sebastian Schlag 21:1–21:19

Minimum Spanning Tree under Explorable Uncertainty in Theory and Experiments
Jacob Focke, Nicole Megow, and Julie Meißner 22:1–22:14

Faster Betweenness Centrality Updates in Evolving Networks
Elisabetta Bergamini, Henning Meyerhenke, Mark Ortmann, and Arie Slobbe 23:1–23:16

Fast Deterministic Selection
Andrei Alexandrescu 24:1–24:19

Fast and Scalable Minimal Perfect Hashing for Massive Key Sets
Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo 25:1–25:16

Generating Practical Random Hyperbolic Graphs in Near-Linear Time and with Sub-Linear Memory
Manuel Penschuck 26:1–26:21

Incremental Low-High Orders of Directed Graphs and Applications
Loukas Georgiadis, Konstantinos Giannis, Aikaterini Karanasiou, and Luigi Laura 27:1–27:21

Jdrasil: A Modular Library for Computing Tree Decompositions
Max Bannach, Sebastian Berndt, and Thorsten Ehlers 28:1–28:21

On the Separation of Topology-Free Rank Inequalities for the Max Stable Set Problem
Stefano Coniglio and Stefano Gualandi 29:1–29:13

Graph Partitioning with Acyclicity Constraints
Orlando Moreira, Merten Popp, and Christian Schulz 30:1–30:15

Bilevel Programming Approaches to the Computation of Optimistic and Pessimistic Single-Leader-Multi-Follower Equilibria
Nicola Basilico, Stefano Coniglio, Nicola Gatti, and Alberto Marchesi 31:1–31:14

The Impact of Landscape Sparsification on Modelling and Analysis of the Invasion Process
Daniyah A. Aloqalaa, Jenny A. Hodgson, and Prudence W. H. Wong 32:1–32:16

Ad-Hoc Affectance-selective Families for Layer Dissemination
Harshita Kudaravalli and Miguel A. Mosteiro 33:1–33:16

Preface

This volume contains papers presented at the 16th International Symposium on Experimental Algorithms (SEA 2017), held June 21–23, 2017, in London, UK.

SEA 2017 continued a now well-established tradition of encouraging high-quality research in experimental computer science and algorithm engineering, providing an opportunity to bring together specialists and young researchers working in the area. The SEA conference series grew out of a seven-year history of the Workshop on Experimental Algorithms (WEA). Previous WEA and SEA meetings have been held in Latvia, Switzerland, Brazil, Greece, Spain, Italy, USA, Germany, Italy, France, and Denmark.

We solicited papers in the broad area of experimental algorithmics, with the Program Committee deciding to accept 30 papers, out of a total of 68 submissions.

Each submission received at least three reviews. Papers were submitted and reviewed using the EasyChair online system. Authors of accepted papers come from 14 countries, across five continents (Asia, Australia, Europe, North America, South America).

The scientific program included three invited lectures, given by:

- Graham Cormode on “Engineering Streaming Algorithms”;
- Martin Farach-Colton on “Dictionaries Revisited”;
- Ruth Misener on “Designing Energy-Efficient Heat Recovery Networks using Mixed-Integer Nonlinear Optimisation”.

We thank the invited speakers for accepting our invitation and for their excellent presentations at the conference.

We thank all authors who submitted their work for consideration at SEA 2017.

We wish to thank the Program Committee and the external reviewers, whose thorough and timely reviews helped us select the presented papers. The success of the scientific program is due to their hard work. We also thank the SEA steering committee for giving us the opportunity to host SEA 2017.

SEA 2017 was organized by the Department of Informatics at King’s College London, whose administrative and financial support we gratefully acknowledge.

London
June 2017

Costas S. Iliopoulos
Solon P. Pissis
Simon J. Puglisi
Rajeev Raman

Program Committee

Maike Buchin	Ruhr University Bochum, Germany
Christina Burt	University of Melbourne, Australia
Sandor Fekete	TU Braunschweig, Germany
Irene Finocchi	University of Rome - La Sapienza, Italy
Ambros Gleixner	Zuse Institute Berlin, Germany
Dominik Kempa	University of Helsinki, Finland
Nicole Megow	University of Bremen, Germany
Ulrich Meyer	Goethe-Universitat Frankfurt am Main, Germany
Shin-Ichi Minato	Hokkaido University, Japan
Petra Mutzel	Technical University of Dortmund, Germany
Gonzalo Navarro	University of Chile, Chile
Giuseppe Ottaviano	Facebook, USA
Panos Pardalos	University of Florida, USA
Solon P. Pissis (Chair)	King's College London, UK
Simon J. Puglisi (Chair)	University of Helsinki, Finland
Rajeev Raman (Chair)	University of Leicester, UK
Barna Saha	University of Massachusetts Amherst, USA
Alessandra Sala	Nokia Bell Labs, Ireland
Sabine Storandt	University of Wuerzburg, Germany
Rossano Venturini	University of Pisa, Italy
Dorothea Wagner	Karlsruhe Institute of Technology, Germany
Renato Werneck	Amazon, USA
Christos Zaroliagis	University of Patras, Greece

External Reviewers

Ajwani, Deepak
Alzamel, Mai
Arroyuelo, Diego
Barth, Lukas
Behdju, Mahyar
Belazzougui, Djamal
Beller, Timo
Berg, Jeremias
Bollhöfer, Matthias
Brinkjost, Tobias
Brückner, Guido
Buchhold, Valentin
Charalampopoulos, Panagiotis
Chavez, Edgar
Choudhary, Keerti
Cseh, Ágnes
D'Andreagiovanni, Fabio
Droschinsky, Andre
Erlebach, Thomas
Fariña, Antonio
Fleischman, Daniel
Fuentes, Jose
Funke, Stefan
Galhotra, Sainyam
Gamrath, Inken
Gog, Simon
Hackfeld, Jan
Hamann, Michael
Heliou, Alice
Herrera, Gioconda
Huang, Chien-Chung
Jabrayilov, Adalat
Karrenbauer, Andreas
Konow, Roberto
Konstantopoulos, Charalampos
Kontogiannis, Spyros
Kriege, Nils
Krinninger, Sebastian
Krupke, Dominik
Kurpicz, Florian
Kurz, Denis
Köppl, Dominik
Lall, Ashwin
Li, Jian
Lübbecke, Marco
Mallozzi, Lina
Margellos, Kostas
Matuschke, Jannik
Mihalák, Matúš
Moreno-Centeno, Erick
Mömke, Tobias
Müller, Benjamin
Nicholson, Patrick K.
Niknejad, Amir
Papagelis, Manos
Penschuck, Manuel
Petri, Matthias
Piperno, Adolfo
Pothitos, Nikolaos
Radermacher, Marcel
Rice, Michael
Schickedanz, Alexander
Schäfer, Till
Serrano, Felipe
Shinano, Yuji
Sorrentino, Francesco
Stiller, Sebastian
Strasser, Ben
Tesch, Alexander
Tischler, German
van der Grinten, Alexander
van der Zanden, Tom
Veith, David
Vitaletti, Andrea
von Looz, Moritz
Välimäki, Niko
Wild, Sebastian
Witzig, Jakob
Zanotto, Leandro
Zey, Bernd
Zündorf, Tobias
Çela, Eranda

16th Symposium on Experimental Algorithms.
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Designing Energy-Efficient Heat Recovery Networks using Mixed-Integer Nonlinear Optimisation

Radu Baltean-Lugojan[1], Christodoulos A. Floudas[2], Ruth Misener[3], and Miten Mistry[4]

1 **Department of Computing, Imperial College London, South Kensington, UK**
rb2309@imperial.ac.uk
2 **Texas A & M Energy Institute, Texas A & M University, College Station, TX, USA**
floudas@tamu.edu
3 **Department of Computing, Imperial College London, South Kensington, UK**
r.misener@imperial.ac.uk
4 **Department of Computing, Imperial College London, South Kensington, UK**
miten.mistry11@imperial.ac.uk

Abstract

Many industrial processes involve heating and cooling liquids: a quarter of the EU 2012 energy consumption came from industry and industry uses 73% of this energy on heating and cooling. We discuss mixed-integer nonlinear optimisation and its applications to energy efficiency. Our particular emphasis is on algorithms and solution techniques enabling optimisation for large-scale industrial networks.

As a first application, optimising heat exchangers networks may increase efficiency in industrial plants. We develop deterministic global optimisation algorithms for a mixed-integer nonlinear optimisation model that simultaneously incorporates utility cost, equipment area, and hot/cold stream matches. We automatically recognise and exploit special mathematical structures common in heat recovery. We also computationally demonstrate the impact on the global optimisation solver ANTIGONE and benchmark large-scale test cases against heuristic approaches.

As a second application, we discuss special structure in nonconvex quadratically-constrained optimisation problems, particularly through the lens of stream mixing and intermediate blending on process systems engineering networks. We take a parametric approach to uncovering topological structure and sparsity of the standard pooling problem in its p-formulation. We show that the sparse patterns of active topological structure are associated with a piecewise objective function. Finally, the presentation explains the conditions under which sparsity vanishes and where the combinatorial complexity emerges to cross over the P/NP boundary. We formally present the results obtained and their derivations for various specialised instances.

1998 ACM Subject Classification G.1.6 Optimization

Keywords and phrases Heat exchanger network, Mixed-integer nonlinear optimisation, Log mean temperature difference, Deterministic global optimisation

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.1

Category Invited Talk

© Radu Baltean-Lugojan, Christodoulos A. Floudas, Ruth Misener, and Miten Mistry;
licensed under Creative Commons License CC-BY

16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Dictionaries Revisited

Martin Farach-Colton

Rutgers University, Piscataway, NJ, USA
martin@farach-colton.com

Abstract

Dictionaries are probably the most well studied class of data structures. A dictionary supports insertions, deletions, membership queries, and usually successor, predecessor, and extract-min. Given their centrality to both the theory and practice of data structures, surprisingly basic questions about them remain unsolved and sometimes even unposed. This talk focuses on questions that arise from the disparity between the way large-scale dictionaries are analyzed and the way they are used in practice.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases B^{ϵ}-trees, file system, write optimization

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.2

Category Invited Talk

© Martin Farach-Colton;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 2; pp. 2:1–2:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Engineering Streaming Algorithms

Graham Cormode

Department of Computer Science, Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, Coventry, UK
`g.cormode@warwick.ac.uk`

Abstract

Streaming algorithms must process a large quantity of small updates quickly to allow queries about the input to be answered from a small summary. Initial work on streaming algorithms laid out theoretical results, and subsequent efforts have involved engineering these for practical use. Informed by experiments, streaming algorithms have been widely implemented and used in practice. This talk will survey this line of work, and identify some lessons learned.

1998 ACM Subject Classification H.2.8 [Database Management] Database Applications, Data mining, F.2.2 [Analysis of Algorithms and Problem Complexity] Nonnumerical Algorithms and Problems

Keywords and phrases Data stream algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.3

Category Invited Talk

© Graham Cormode;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 3; pp. 3:1–3:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Better Process Mapping and Sparse Quadratic Assignment*

Christian Schulz[1] and Jesper Larsson Träff[2]

1 **Karlsruhe Institute of Technology, Karlsruhe, Germany; and University of Vienna, Vienna, Austria**
`christian.schulz@kit.edu, christian.schulz@univie.ac.at`

2 **TU Wien, Vienna, Austria**
`traff@par.tuwien.ac.at`

Abstract

Communication and topology aware process mapping is a powerful approach to reduce communication time in parallel applications with known communication patterns on large, distributed memory systems. We address the problem as a quadratic assignment problem (QAP), and present algorithms to construct initial mappings of processes to processors as well as fast local search algorithms to further improve the mappings. By exploiting assumptions that typically hold for applications and modern supercomputer systems such as sparse communication patterns and hierarchically organized communication systems, we arrive at significantly more powerful algorithms for these special QAPs. Our multilevel construction algorithms employ recently developed, perfectly balanced graph partitioning techniques and excessively exploit the given communication system hierarchy. We present improvements to a local search algorithm of Brandfass et al. (2013), and decrease the running time by reducing the time needed to perform swaps in the assignment as well as by carefully constraining local search neighborhoods. Experiments indicate that our algorithms not only dramatically speed up local search, but due to the multilevel approach also find much better solutions in practice.

1998 ACM Subject Classification G.2.2 [Graph Theory] Graph Algorithms, G.4 [Mathematical Software] Algorithm Design and Analysis

Keywords and phrases rank reordering, graph algorithms, process mapping, graph partitioning

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.4

1 Introduction

Communication performance between processes in high-performance systems depends on many factors. For example, communication is typically faster if communicating processes are located on the same processor node compared to the cases where processes reside on different nodes. This becomes even more pronounced for large supercomputer systems where processors are hierarchically organized into, e. g., islands, racks, nodes, processors, cores with corresponding communication links of similar quality. Given the communication pattern between processes and a hardware topology description that reflects the quality of the communication links, one hence seeks to find a good mapping of processes onto processors such that pairs of processes exchanging large amounts of data are located closely.

Such a mapping can be computed by solving a corresponding quadratic assignment problem (QAP) which is a hard optimization problem. Sahni and Gonzalez [26] have shown

* This work was partially supported by DFG grants SA 933/11-1.

© Christian Schulz and Jesper Larsson Träff;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 4; pp. 4:1–4:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

QAP to be strongly NP-hard and, unless P=NP, admitting no polynomial-time constant factor approximation algorithm. In addition, there are no algorithms that can solve meaningful instances with $n > 20$ to optimality in a reasonable amount of time [9]. Hence, heuristic algorithms are necessary in order to solve large scale instances. Multiple heuristics have been proposed to tackle real world instances [7, 17, 23]. We present more details in Section 3.

In this work, we make two important assumptions that are typically valid for modern supercomputers and the applications that run on those. First, communication patterns are almost always sparse since not all processes have to communicate with each other. This is especially true for large scale scientific simulations in which the underlying models of computation and communication are already sparse, see, e. g., [10, 14, 28]. To efficiently parallelize the simulation one normally employs graph partitioning techniques which then in turn yield a sparse communication pattern between the processes. Second, we assume that the hardware communication topology under consideration is hierarchical with communication links on the same level in the hierarchy having the same communication speed. This is typically observed in current high-performance systems, e. g., SuperMUC[1].

Using these assumptions, we derive algorithms that are able to create high quality mappings, as well as faster local search algorithms for improving assignments. Overall, our algorithms are able to compute better solutions than other recent heuristics for the problem. Improving the (practical) complexity of such algorithms is highly important, since the number of cores available in supercomputers is still increasing dramatically. The rest of this paper is organized as follows. In Section 2, we introduce basic concepts and describe relevant related work, such as the algorithm of Brandfass et al. [7], in more detail. We present our main contributions in Section 3. We implemented the techniques presented here in the graph partitioning framework KaHIP [27] (Karlsruhe High Quality Graph Partitioning). A summary of extensive experiments to evaluate algorithm performance is presented in Section 4. Experiments indicate that our algorithm not only drastically speeds up local search, but due to the multilevel approach that employs recently developed high quality partitioning techniques also finds better solutions in practice.

2 Preliminaries

The total communication requirement between the set of processes in (some section of) an application can be modeled by a weighted communication graph. The underlying hardware topology can likewise be modeled by a weighted graph, but since the graph is complete (any physical processor can communicate with any other physical processor through the underlying networks), we represent it by a topology cost matrix which can for instance reflect the costs of routing along shortest (cheapest) paths between processes. Our abstract problem is to embed the communication graph onto the topology graph under optimization criteria that we explain below. We assume that the number of nodes in host and topology graphs are the same. Unless otherwise mentioned, a processing element (PE) typically represents a core of a machine.

Basic Concepts. In the following, we consider an undirected graph $G = (V = \{0, \dots, n-1\}, E)$ with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i. e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. We let $\Gamma(v) := \{u : \{v, u\} \in E\}$ denote the neighbors of a node v. A graph $S = (V', E')$ is said

[1] Leibniz Supercomputing Centre, Gauss Centre for Supercomputing e.V.

to be a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. We call S an *induced* subgraph when $E' = E \cap (V' \times V')$.

Throughout the paper, $\mathcal{C} \in \mathbb{R}^{n \times n}$ denotes the communication matrix and $\mathcal{D} \in \mathbb{R}^{n \times n}$ the hardware topology matrix or distance matrix. More precisely, $\mathcal{C}_{i,j}$ describes the amount of communication that has to be done between process i and j, and $\mathcal{D}_{i,j}$ represents the weighted distance between PE i and PE j. That is, the cost for communicating the amount $\mathcal{C}_{i,j}$ between processors i and j is $\mathcal{C}_{i,j}\mathcal{D}_{i,j}$. We follow Brandfass et al. [7] and others, and model the embedding problem as a quadratic assignment problem (QAP): Find a one-to-one mapping Π of processes to PEs which minimizes the overall communication cost. More precisely, we want to minimize $J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{i,j} \mathcal{C}_{\Pi(i),\Pi(j)} \mathcal{D}_{i,j}$ where the sum is over all PE pairs and $k = \Pi(i)$ means that process k is assigned to PE i. Note that searching for the inverse permutation instead, i. e., assigning process i to PE $\Pi^{-1}(i)$, results in the same assignment problem as Π is a one-to-one mapping. Throughout this work, we assume that $\mathcal{C}$ and $\mathcal{D}$ are symmetric – otherwise one can create equivalent QAP problems with symmetric inputs [7]. In this paper, we focus on *sparse* communication patterns, and therefore do not want to store the complete communication matrix but instead represent it more efficiently as a graph. On the other hand, typical system topologies feature a hierarchy that we can exploit. Hierarchy information, and in general $\mathcal{D}$, is given implicitly and can be queried, and therefore does not have to be stored explicitly.

Graph partitioning is a key component in our algorithms to find initial solutions. The *graph partitioning problem* looks for *blocks* of nodes $V_1, \ldots, V_k$ that partition V, i. e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in 1..k : c(V_i) \leq L_{\max} := (1 + \epsilon)\lceil c(V)/k \rceil$ for some parameter ϵ. In the *perfectly balanced case* the imbalance parameter ϵ is set to zero, i. e., no deviation from the average is allowed. One commonly used objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$.

Related Work. There has been a *huge* amount of research on GP, and we refer the reader to [4, 8] for extensive material and references. All general-purpose methods that work well on large real-world graphs are based on the multilevel principle. The basic idea can be traced back to multigrid solvers for systems of linear equations [31]. Well-known MGP software packages include Jostle [34], Metis [19], and Scotch [25]. Jostle contains algorithms to compute processor assignments in scientific simulations. Jostle integrates local search into a multi-level method to partition the model of computation and communication. To do so, they solve the problem on the coarsest level and afterwards perform refinement that takes the user supplied network communication model into account. Scotch performs dual recursive bipartitioning to perform this task. There is likewise a large literature on process mapping [6, 24], often in the context of scientific applications using MPI [22] (Message-Passing Interface). Bokhari shows that the mapping problem is equivalent to the graph isomorphism problem [5]. Hatazaki [16] was among the first authors to propose graph partitioning to solve the process mapping problem for unweighted process topologies in the specific context of MPI. Träff [32] used a similar approach, and gave one of the first non-trivial implementations for the NEC vector supercomputers. Mercier and Clet-Ortega and later Jeannot [20, 21] simplify the mapping problem by only considering the topology inside the compute nodes themselves and ignoring the topology of the network. Multiple placement policies are investigated to enhance overall system performance. Yu et al. [35] discuss and implement embedding heuristics for the BlueGene $3d$ torus systems. Hoefler and Snir [18] optimize instead the congestion of the mapping, show that this problem is NP-complete, and give a corresponding heuristic with an experimental evaluation based on application data from the Florida Sparse Matrix Collection.

Routing aware mapping heuristics taking the hierarchy of specific hardware topologies into account were discussed in [1]. A resource-aware graph partitioning framework has been proposed by Chan et al. [11]. Vogelstein et al. [33] concentrate on solving general quadratic assignment and graph matching problems. They propose a gradient based heuristic that involves solving assignment problems and give experimental evidence for better solution quality and speed compared to certain other heuristics. The worst-case complexity of their approach is $O(n^3)$ steps.

Detailed Related Work. We now discuss related work by Müller-Merbach [23], Heider [17] and Brandfass et al. [7] as well as Glantz et al. [15] in greater detail since our work either makes use of the tools proposed by those authors or because we compare against their results. Müller-Merbach [23] proposes a greedy construction method to obtain an initial permutation for the QAP. The method roughly works as follows: Initially compute the total communication volume for each processor and also the total distance for each core. Note that this corresponds to the weighted degrees of the vertices in the communication and distance models, respectively. Afterwards, the process with the largest communication volume is assigned to the core with the smallest total distance. To build a complete assignment, the algorithm proceeds by looking at unassigned vertices and cores. For each of the unassigned processes the communication load to already assigned vertices is computed. For each core, the total distance to already assigned cores is computed. The process with the largest communication sum is assigned to the core with the smallest distance sum. Glantz et al. [15] note that the algorithm does not link the choices for the vertices and cores and propose a modification of this algorithm called *GreedyAllC* (the best algorithm in [15]). GreedyAllC links the mapping choices by scaling the distance with the amount of communication to be done. The algorithm has the same asymptotic complexity and memory requirements as the algorithm by Müller-Merbach. We also compare our proposed methods against GreedyAllC in Section 4.

Heider [17] proposes a method to improve an already given permutation/mapping. The method repeatedly tries to perform swaps in the assignment. To do so, the author defines a pair-exchange neighborhood $N(\Pi)$ that contains all permutations that can be reached by swapping two elements in Π. Here, swapping two elements means that $\Pi(i)$ will be assigned to processor j and $\Pi(j)$ will be assigned to processor i after the swap is done. The algorithm then looks at the neighborhood in a cyclic manner. More precisely, in each step the current pair (i, j) is updated to $(i, j+1)$ if $j < n$, to $(i+1, i+2)$ if $j = n$ and $i < n-1$, and lastly to $(1, 2)$ if $j = n$ and $i = n-1$. A swap is performed if it yields positive gain, i. e., the swap reduces the objective. The overall runtime of the algorithm is $O(n^3)$. We denote the search space with N^2. To reduce the runtime, Brandfass et al. [7] introduce a couple of modifications. First of all, only symmetric inputs are considered. If the input is not symmetric, the input is substituted by a symmetric one such that the output of the algorithm remains the same. Second, pairs (i, j) for which the objective cannot change, are not considered. For example, if two processes reside on the same compute node, swapping them will not change the objective. Lastly, the authors partition the neighborhood search space into s consecutive index blocks and only perform swaps inside those blocks. This reduces the number of possible pairs from $O(n^2)$ to $O(ns)$ overall pairs. We denote the search space with $\mathcal{N}_p$ (*pruned neighborhood*). In addition, instead of starting from the identity permutation, the authors use the method of Müller-Merbach [23] to compute an initial solution. This improves runtime of the local search approach as well as the objective of the solution.

3 Rank Reordering Algorithms

We now present our main contributions and techniques. This includes algorithms to compute initial solutions, speeding up the local search algorithms for sparse communication patterns and defining new search spaces for the local search algorithm. Throughout this section, we assume that the input communication matrix is already given as a graph $G_\mathcal{C}$, i. e., no conversion of the matrix into a graph is necessary. More precisely, the graph representation is defined as $G_\mathcal{C} := (\{1, \dots, n\}, E[\mathcal{C}])$ where $E[\mathcal{C}] := \{(u, v) \mid \mathcal{C}_{u,v} \neq 0\}$. In other words, $E[\mathcal{C}]$ is the edge set of the processes that need to communicate with each other. Note that the set contains forward and backward edges, and that the weights of the edges in the graph correspond to the entries in the matrix $\mathcal{C}$.

3.1 Initial Solutions

We propose two strategies exploiting the hierarchy. Intuitively, we want to identify subgraphs in the communication graph of processes that have to communicate much with each other and then place such processes closely, i. e., on the same node, same rack and so forth. In the following, we assume a homogeneous hierarchy of the supercomputer, but our algorithms can be extended to heterogeneous hierarchies in a straightforward way. Let $\mathcal{S} = a_1, a_2, ..., a_k$ be a sequence describing the hierarchy of the supercomputer (with $n = \Pi_i a_i$). The sequence should be interpreted as each processor having a_1 cores, each node a_2 processors, each rack a_3 nodes, ... We propose two algorithms to compute initial mappings, a top down and a bottom up approach. The first one, *top down*, splits the communication graph recursively and the second one, builds a hierarchy *bottom up*.

The *top down approach* starts by computing a *perfectly balanced* partition of $G_\mathcal{C}$ into a_k blocks each having n/a_k vertices (processes). The partitioning task is done using the techniques provided by Sanders and Schulz [27] which provide high quality partitions and guarantee that each block of the output partition has the specified amount of vertices. In principle, the nodes of each block will be assigned completely to one of the a_k system entities. Each of the system entities provides precisely n/a_k PEs. We then proceed recursively and partition each subgraph induced by a block into a_{k-1} blocks and so forth. The recursion stops as soon as the subgraphs have only a_1 vertices left. In the base case, we assign processes to permutation ranks.

The *bottom up approach* proceeds in the opposite order of the hierarchy. That means the communication graph $G_\mathcal{C}$ is split first into $k = n/a_1$ blocks with precisely a_1 vertices each. Again, this is done using the perfectly balanced partitioning techniques mentioned above. Each block will later on be assigned to a unique system entity that is able to host a_1 processes, i. e., a node having a_1 cores. Then each of the blocks is contracted and we partition the contracted graph and so forth. In this case, if replacing edges of the form $\{u, w\}, \{v, w\}$ would generate two parallel edges $\{x, w\}$, we insert a single edge with $\mathcal{C}'_{x,w} = \mathcal{C}_{u,w} + \mathcal{C}_{v,w}$. This way, the correct sum of the distances are accounted for in later stages of the algorithm. The recursion stops as soon as the last hierarchy stage is reached, i. e., the last graph with n' vertices has been partitioned into n'/a_k vertices with a_k vertices each. Recall that vertices in the same block will be assigned to a specified subpart of the system. In this case, a vertex in the graph on the last level of the recursion represents a whole set of tasks with the property that the sum of the vertex weights of each block is precisely the amount of PEs that are present in the subsystem that they are assigned to. We then backtrack the recursion to construct the final mapping.

3.2 Faster Swapping

Initially computing as well as recomputing the objective function after a swap is performed is an expensive step in the algorithm of Brandfass et al. [7]. In their work, both the communication pattern as well as the distances between the PEs are given as complete matrices. These matrices have a quadratic number of elements and hence the initial computation of the objective function costs $O(n^2)$ time. After a swap is performed, Brandfass et al. update the objective using the objective function value before the swap. This is done by looking at all elements in the corresponding columns of the communication and distance matrix. Overall, an update step in their algorithm takes $O(n)$ time which is clearly a bottleneck for sparse communication patterns. We now describe how we speed up the initial computation as well as the update of the objective. As a first step, we rewrite the objective to work with the inverse of the permutation:

$$\begin{aligned} J(\mathcal{C}, \mathcal{D}, \Pi) &= \sum_{i,j} \mathcal{C}_{\Pi(i),\Pi(j)} \mathcal{D}_{i,j} \\ &= \sum_{u,v} \mathcal{C}_{u,v} \mathcal{D}_{\Pi^{-1}(u),\Pi^{-1}(v)} \end{aligned}$$

with the interpretation that task u is assigned to PE $\Pi^{-1}(u)$. This makes it easier to work with the graph representation of the communication matrix. We rewrite the objective to work with the graph representation instead of the complete communication pattern matrix $\mathcal{C}$:

$$J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{(u,v) \in E[\mathcal{C}]} \mathcal{C}_{u,v} \mathcal{D}_{\Pi^{-1}(u),\Pi^{-1}(v)}.$$

The first observation is that given an initial mapping, we can compute the initial objective in $O(n + m)$ time which is better for sparse graphs. Our next goal is to make the update of the objective fast after a swap has been performed. To do so, let $\Gamma_{\Pi^{-1}}(u) := \sum_{v \in N(u)} \mathcal{C}_{u,v} \mathcal{D}_{\Pi^{-1}(u),\Pi^{-1}(v)}$ be the contribution to the objective of a single vertex u given the current mapping. Note that by using $\Gamma_{\Pi^{-1}}$, we can again rewrite the objective $J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{u \in V} \Gamma_{\Pi^{-1}}(u)$. Throughout the algorithm, the vertex contributions Γ are always kept up to date. Additionally, it is quite easy to see that performing a swap in the assignment only affects the nodes that are swapped themselves as well as their neighborhood in the communication graph. Hence, we only need to update the node contributions of those nodes and can update the objective accordingly. We update the node contributions as follows: Let u and v be the vertices to be swapped in their assignment Π^{-1}. We start by subtracting the node contributions of all affected nodes from the objective. Before we perform the swap, we iterate over the neighbors of u and v and subtract the contribution induced by the edge connecting the neighbor from its Γ value. We then set the node contributions of u and v to zero and perform the swap. Now we again iterate over all neighbors, basically recomputing the node contributions of u and v, and at the same time adding the new contribution induced by the edge connecting the neighbor to its Γ value. As a last step, we add the new node contributions of all affected nodes from the objective. Overall, this takes $O(d_u + d_v)$ time where d_u and d_v are the degrees of the vertices u and v in the communication graph.

3.3 Alternative Local Search Spaces

We now define swapping neighborhoods using the communication graph $G_{\mathcal{C}}$. In the simplest version, assignments are only allowed to be swapped if the processes are connected by an edge in the communication graph, i. e., the processes have to communicate with each other.

We denote this neighborhood with $N_{\mathcal{C}}$. The size of the search space is $O(m)$ since it contains exactly m pairs that may be swapped. Swaps are performed in random order. Local search terminates after m unsuccessful swaps, i. e., all pairs have been tried and no swap resulted in a gain in the objective. Note that this approach assumes that swaps with positive gain are close in terms of graph theoretic distance in the communication graph. We also define augmented neighborhoods in which swaps are allowed if two processes have distance less than d in the communication graph. We denote this neighborhood by $N_{\mathcal{C}}^d$. Note that this creates a sequence of neighborhoods increasing in size $N_{\mathcal{C}} \subseteq N_{\mathcal{C}}^2 \subseteq \ldots \subseteq N_{\mathcal{C}}^n = N^2$ where N^2 is the largest neighborhood used by Brandfass et al. [7] (see Section 2). Our experimental section shows that performing swaps with small graph theoretic distance in the communication graph is sufficient to obtain good solutions.

3.4 Miscellanea

Constant Time Distance Oracle. Storing the complete distance matrix requires $O(n^2)$ space. However, due to the problem structure it is not necessary to store the complete matrix. Instead one can build an interval tree over the given PE's describing the hierarchy. The distance of two PEs can then be found by finding the lowest common ancestor in the tree. Such a query can be answered in constant time by investing $O(n)$ preprocessing time [3].

We can use a simpler approach that obtains the distance of two PEs by a few, simple division operations. More precisely, for a hierarchy $\mathcal{S} = a_1, a_2, ..., a_k$ we initially build an array describing the sizes of the intervals on the different levels of the hierarchy. A query then proceeds to scan the implicitly given intervals from top to bottom until the PEs are not on the same subsystem. We then return the corresponding distance.

4 Experiments

Methodology. We have implemented the algorithm described above within the KaHIP framework using C++ and compiled all algorithms using gcc 4.6.3 with full optimization's turned on (-O3 flag). We integrated our algorithms in KaHIP v1.00 graph partitioning framework where the mapping codes are used as post-processing as well as in a separate release VieM [29] (Vienna Mapping and Sparse Quadratic Assignment) to make the mapping codes themselves available to a broader audience. They will integrated into that framework and also released separately. The codes of Brandfass et al. [7] could not be made available to us, so that we implemented those algorithms in our framework as well. Our implementation also uses the sparse representation of the communication pattern. GreedyAllC [15] has been kindly provided by the authors. We also compare against the dual recursive bisection codes of Hofler and Snir [18] (LibTopoMap). Our experiments evaluate the objective of the quadratic assignment problem as well as the running time necessary to compute the solution. To keep the evaluation simple, we use mostly one system hierarchy configuration $\mathcal{D}$. We perform ten repetitions of each algorithm using different random seeds for initialization. Unless otherwise mentioned, we use the geometric mean when reporting averages in order to give every instance the same influence on the *final score*. The system we are using to compute solutions has four Octa-core Intel Xeon E5-4640 processors (32 cores) which run at a clock speed of 2.4 GHz. It has 512 GB local memory.

Instances. We use graphs from various sources to test our algorithm. In Section 4.1, we use these graphs as input to a partitioning algorithm that partitions them into a given number of blocks and then computes the communication graph $\mathcal{C}$ which is the input to our mapping

Table 1 Average running time and average speedup of local search for pruned search space N_p. Here, $\overline{m/n}$ is the average density of the instances, t_{LS} the average running time of the algorithm using slow gain computations and t_{fastLS} the average running time using fast gain computations.

n	$\overline{m/n}$	$t_{\mathrm{LS}}[s]$	$t_{\mathrm{fastLS}}[\mathrm{s}]$	speedup
64	6.7	0.016	0.003	5.3
128	7.3	0.064	0.006	10.7
256	7.9	0.268	0.014	19.1
512	8.3	1.073	0.029	37.0
1K	8.8	4.263	0.059	72.3
2K	9.2	17.083	0.124	137.8
4K	9.7	68.360	0.260	262.9
8K	10.3	268.907	0.540	498.0
16K	11.2	1 075.107	1.158	928.4
32K	12.5	4 348.374	2.472	1 759.1

algorithms. We use the largest six graphs from Chris Walshaw's benchmark archive [30]. Graphs derived from sparse matrices have been taken from the Florida Sparse Matrix Collection [12]. We also use graphs from the 10th DIMACS Implementation Challenge [2] website. Here, `rggX` is a *random geometric graph* with 2^X nodes where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. The graph `delX` is a Delaunay triangulation of 2^X random points in the unit square. The graphs `af_shell9`, `thermal2`, and `nlr` are from the matrix and the numeric section of the DIMACS benchmark set. The graphs `europe` and `deu` are large road networks of Europe and Germany taken from [13]. Basic properties of the graphs under consideration can be found in Table A.3.

4.1 Sparse Quadratic Assignment Problem

In this section, we look at the impact of the various algorithmic components that we presented throughout the paper. In general, we use a hierarchy $\mathcal{S} = a_1, \ldots, a_k$ describing the system hierarchy and communication parameters $D = d_1, \ldots, d_k$ describing the distances between various cores in the subsystems. More precisely, d_i describes the distance of two cores that are in the same subsystems for $i' < i$, and in different subsystems for $i' \geq i$. The total number of cores is given by $n = \prod_i a_i$. Here, we focus on two different system configurations to keep the evaluation simple. Our process in this section is as follows: Take the input graph, partition it into n blocks using the fast configuration of KaHIP, compute the communication graph induced by that (vertices represent blocks, edges are induced by connectivity between blocks, edge cut between two blocks is used as communication volume) and then compute the mapping of the communication graph to the specified system.

4.1.1 Speed-Up of Local Search

We now take the algorithm configurations initially used by Brandfass et al. [7] and investigate the impact of our faster local search algorithms. The configurations are as follows: Use the greedy growing algorithm by Müller-Merbach (as described in Section 2) to provide initial solutions and use the pruned local search neighborhood N_p by Brandfass et al. [7] (see Section 2 for details). We run two configurations: One in which computing the gain takes linear time (the old algorithm) and one with our improved algorithm. In this experiment,

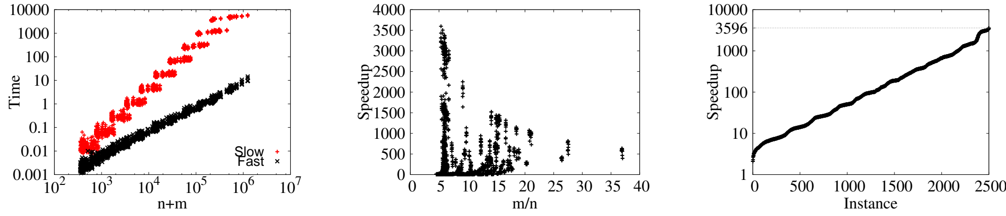


Figure 1 From left to right: Time of local search for both configurations (slow and fast), algorithmic speedup as a function of graph density, algorithmic speedup for the different instances.

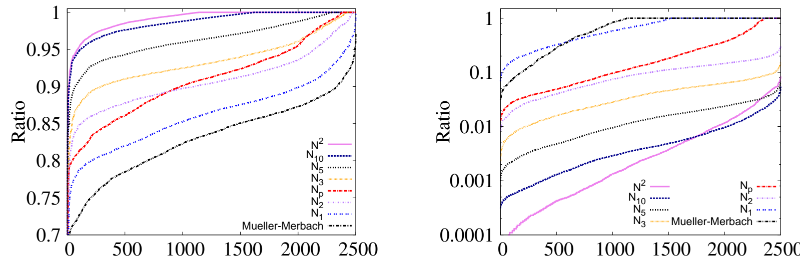


Figure 2 Left/Right: performance plot with respect to solution quality/running time for different local search algorithms.

we use $S = 4:16:k$, $D = 1:10:100$ with $k = 2^i$, $i \in \{1, ..., 9\}$. Note that the objective of the computed solutions by the algorithm using faster gain computations is precisely the same as their counter part, hence we *do not* report the value of the objective in this section. The results of the experiments are summarized in Figure 1 and Table 1. First, we observe that our new algorithm is *always* faster than the old algorithm. This is expected since the models of computation and communication that are mapped are indeed sparse. Table 1 shows that our fast local search algorithm scales almost linearly in n while the algorithm not using fast gain computations shows quadratic scaling behaviour. The table also already shows a dependency of our algorithm on the density of the instances. This is due to the fact that the gain computation depends on the degrees of the vertices in the communication graph and is in alignment with our theoretical analysis. The expected dependency on the density of the instances can also be seen more clearly in Figure 1. The smallest algorithmic speedup obtained in this experiment is two and the largest speedup is approximately 3 596. We conclude that exploiting the sparsity of the application can improve the running time of local search significantly. From now on, we now always use fast gain computations.

4.1.2 Local Search Neighborhoods

In this section, we look at the influence of local search neighborhoods on final solution quality. The base configuration used here employs the greedy growing algorithm by Müller-Merbach for initialization. Afterwards local search is done using the specified local search neighborhood, i. e., the quadratic neighborhood N^2, the pruned quadratic neighborhood N_p and the communication graph based neighborhoods $N_d := N_C^d$ for $d \in \{1, 2, 3, 5, 10\}$. Again, we use $S = 4:16:k$, $D = 1:10:100$ with $k = 2^i$, $i \in \{1, ..., 9\}$. To get a visual impression of the solution quality of the different algorithms, Figure 2 presents *performance plots* using all instances. A curve in a performance plot for algorithm X is obtained as

Table 2 Average ratios for solution quality and running time. Baseline denotes the construction heuristic of Müller-Merbach without local search. Algorithms use the baseline algorithm and add local search with the respective local search neighborhood. Comparisons are done against the baseline algorithm. Quality improvements are shown in %.

n	N^2	N_p	N_1	N_2	N_{10}	N^2	N_p	N_1	N_2	N_{10}
	baseline/{baseline+local search} quality improvement [%]					local search/baseline average running time ratios:				
64	17.4	17.4	6.3	13.0	17.2	26.2	27.1	2.6	13.3	44.1
128	16.0	10.9	3.8	8.5	15.4	63.9	25.2	2.7	16.8	92.8
256	17.3	10.0	3.4	8.3	17.3	114.7	18.9	2.5	16.3	149.0
512	17.6	8.9	3.2	8.0	17.5	171.8	11.3	1.8	12.7	190.2
1K	18.8	8.2	3.1	8.2	18.2	259.1	6.8	1.3	10.0	245.1
2K	19.5	8.1	3.1	8.2	19.1	348.2	3.7	0.9	7.0	258.6
4K	20.5	8.0	3.3	8.7	19.8	472.0	2.0	0.6	5.1	231.8
8K	21.6	8.0	3.6	9.4	20.9	728.2	1.0	0.5	4.0	212.0
16K	23.1	8.3	4.2	10.4	22.1	1 030.8	0.6	0.3	2.9	173.6
32K	25.0	9.1	5.4	11.9	23.7	1 220.9	0.3	0.2	2.1	128.2
overall:	19.68	9.69	3.94	9.46	19.12	443.58	9.69	1.34	9.02	172.54

follows: For each instance, we calculate the ratio between the objective or running time obtained by any of the considered algorithms and objective or running time of algorithm X. These values are then sorted. Additionally, we present average ratios of solution quality and running time in Table 2. First, the local search algorithm using the N_1 neighborhood appears to be the fastest algorithm but also the worst in terms of solution quality. Compared to the initial construction heuristic it takes roughly a factor 1.34 in running time while improving solution quality by roughly 4%. With increasing distance d for the local search neighborhood N_d, solutions improve but also the running time increases. As expected, the local search algorithm using the largest local search neighborhood N^2 computes the best solutions. Here, solutions generated by the initial heuristic are improved by roughly 20%. However, this is also the slowest algorithm (a factor 443 slower than the initial construction heuristic). Also note that we are only able to evaluate the performance of the algorithm at that scale due to the fast gain computations introduced in this paper. Additionally, as n increases the algorithm becomes much slower. This is due to the fact that convergence of the algorithm takes more time for larger n. In contrast, the other local search neighborhoods show much better scaling behaviour as expected. The local search neighborhood N_{10} is faster and computes solutions that are only slightly worse than N^2. For example, for $n = 32$K the algorithm using N_{10} is more than a factor nine faster and computes solutions that are only 5.5% worse.

4.1.3 Initial Heuristics and Their Scaling Behaviour

We now evaluate the different heuristics that can be used to create solutions. For the evaluation, we employ the algorithm of Müller-Merbach [23], GreedyAllC [15], LibTopoMap [18] (dual recursive bisectioning), Identity, Random, the Bottom-Up as well as the Top-Down and the Top-Down algorithm combined with local search that uses the N_{10} neighborhood (Top-Down+N_{10}). The problems we look at are defined as $S_1 = 4 : 16 : k$, $D_1 = 1 : 10 : 100$ with $k \in \{1, \ldots, 128\}$. We run the Bottom-Up algorithm only to $k \leq 50$ due to its large running time. Figure 3 shows the average improvement over solutions obtained by the

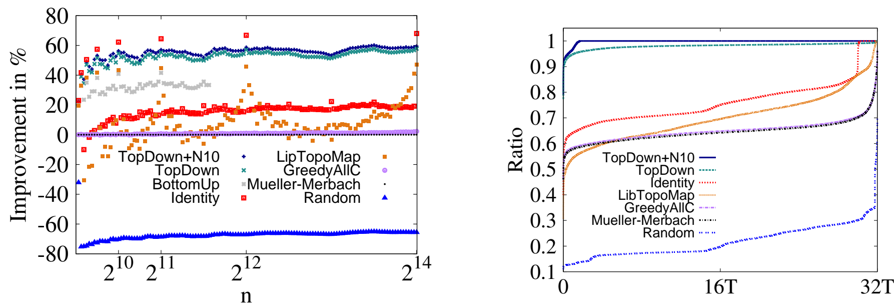


Figure 3 Average improvement in % for different values of n for different algorithms over the algorithm by Müller-Merbach (top) and a performance plot comparing solution quality (bottom).

algorithm of Müller-Merbach and a performance plot for the different algorithms. Indeed, the random mapping algorithms perform worse than the algorithm of Müller-Merbach. On average, the objective computed by the algorithm is 67% worse than the solutions computed by the algorithm of Müller-Merbach. Our Top-Down algorithms yields the best solutions on most of the instances. On average, solutions computed by Top-Down are 52% better than the solutions computed by Müller-Merbach. Adding local search with the N_{10} neighborhood to the algorithm yields additional 5.3% improvement on average. GreedyAllC only improves slightly, i. e., 1% on average, over the algorithm of Müller-Merbach. The identity mapping seems to be the best algorithm for powers of two. This is due to the way the input to the algorithms is constructed, i. e., blocks are initially assigned by KaHIP. To be more precise, KaHIP uses a recursive bisection algorithm on the input graph to compute a model of computation and communication (the input to our mapping algorithms). In each recursion it assigns consecutive blocks to the left side and to the right side. Hence, for powers of two, the identity mapping yields a strategy similar to using recursive bisection on the model to be mapped with good bisections. If the number of elements is not a power of two, then the bisections implied by the identity are not good and hence it performs worse.

LibTopoMap is somewhere in between. It mostly computes better solutions than the greedy algorithms but overall worse solutions than BottomUp and TopDown. On average, solutions are 8% better than the solutions computed by the greedy algorithm of Müller-Merbach. Interestingly, its achieved solution quality is better when the number of vertices in the instances is close to a power of two. This is due to the fact that the algorithm uses dual recursive bisection on the communication and processor graph. However, when the input size is not close to a power of two, there are no good bisections in the processor graph.

In our experiments, Bottom-Up is the slowest algorithm. This is due to the fact that on the coarsest level large partitioning problems have to be solved. The Top-Down algorithm does not have the problem, but is still slower than all other algorithms (except Bottom-Up). On average it is a factor 194 slower than the Müller-Merbach algorithm and a factor 40 slower than GreedyAllC. LibTopoMap is roughly a factor 18 slower than the algorithm of Müller-Merbach. However, the running time of Top-Down is on average only 80% of the time it takes to partition the input graph (using the fast configuration of KaHIP), i. e., the time it takes to create the model which is the input to the mapping algorithms. Adding local search with the N_{10} neighborhood to the algorithm costs additional time, on average 64% of the time it takes to partition the graph. Considering also the high solution quality advantage, we believe that the algorithms are still highly useful in practice.

Scalability. We now scale the problem size to $n = 2^{19}$ processes/cores. We take the largest graph from our benchmark collection rgg24 and create mapping problems defined as $S_1 = 4:16:128:k$, $D_1 = 1:10:100:1000$ with $k \in 2^i, i \in \{1, \ldots, 8\}$. We run Müller-Merbach and the TopDown+N_1 algorithm once. Both algorithms work well on our machine until $i = 4$ ($n = 2^{17}$), at which point there is not sufficient memory available if the implementations use the full distance matrix. Note that the machine has 512GB of memory. Hence, we performed a second run of both algorithms computing distances online (as described in Section 3.4). Note that the version of the Müller-Merbach algorithm is only able to solve larger problem sizes due to both of our changes: the sparse representation of the communication pattern as well as online computation of distances. Computing distances online slows down Müller-Merbach roughly by a factor of five and local search by a factor of three. The running time of TopDown remains the same since it uses the provided hierarchy instead of the distance matrix. In turn the running time advantage of Müller-Merbach also decreases. This is also due to the fact that Müller-Merbachs algorithm is a quadratic time algorithm. For the largest mapping problem ($n = 2^{19}$), the Müller-Merbach algorithm takes a factor 1.64 longer than TopDown. Overall, computing distances online enables a potential user of the algorithms to tackle larger mapping problems.

5 Conclusion

In high performance systems, different cores that are on the same processor usually have the same communication link quality when they communicate with each other, as do cores that are on the same node but not on the same processor and so forth. Using these assumptions, we derived algorithms to create initial mappings as well as faster local search algorithms with alternative local search spaces. Overall, our algorithms drastically speedup local search and are able to compute high quality solutions.

Important future work includes deriving distributed parallel algorithms for the problem. Moreover, we want to investigate algorithms to create a hierarchy of the system if it is not provided as an input to our algorithm. It may be worth to look at more complex local search neighborhoods, e. g., local search spaces that allow to swap whole groups of assignments or allow swapping along cycles in the communication graph. We also want to study the impact of our process mapping on parallel application performance.

References

1. A. H. Abdel-Gawad, M. Thottethodi, and A. Bhatele. RAHTM: Routing Algorithm Aware Hierarchical Task Mapping. In *Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 325–335, 2014.
2. D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.
3. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, volume 1776, pages 88–94. Springer, LNCS, 2000.
4. C. Bichot and P. Siarry, editors. *Graph Partitioning*. Wiley, 2011.
5. S. H. Bokhari. On the mapping problem. *IEEE Trans. Computers*, 30(3):207–214, 1981. URL: `http://dx.doi.org/10.1109/TC.1981.1675756`, `doi:10.1109/TC.1981.1675756`.
6. S. H. Bokhari. Assignment problems in parallel and distributed computing, 2012.
7. B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers & Fluids*, 80:372–380, 2013.

8 A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering – Selected Topics*, to app., *ArXiv:1311.3144*, 2014.
9 R. E. Burkard, E. Cela, P. M. Pardalos, and L. S. Pitsoulis. The quadratic assignment problem. In *Handbook of combinatorial optimization*, pages 1713–1809. Springer, 1998.
10 Ü. V. Çatalyürek and C. Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In *Proc. of the 3rd Int'l Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 1117, pages 75–86. Springer, 1996.
11 Siew Yin Chan, Teck Chaw Ling, and Eric Aubanel. The impact of heterogeneous multi-core clusters on graph partitioning: an empirical study. *Cluster Computing*, 15(3):281–302, 2012. doi:10.1007/s10586-012-0229-4.
12 T. Davis. The University of Florida Sparse Matrix Collection.
13 D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS State-of-the-Art Survey*, pages 117–139. Springer, 2009.
14 J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In *Proc. of Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 818–829. Springer, 2012.
15 R. Glantz, H. Meyerhenke, and A. Noe. Algorithms for mapping parallel processes onto grid and torus architectures. In *23rd Euromicro Int'l Conference on Parallel, Distributed, and Network-Based Processing*, pages 236–243. IEEE Computer Society, 2015. doi:10.1109/PDP.2015.21.
16 T. Hatazaki. Rank reordering strategy for MPI topology creation functions. In *5th European PVM/MPI User's Group Meeting*, volume 1497 of *LNCS*, pages 188–195. Springer, 1998.
17 C. H. Heider. A computationally simplified pair-exchange algorithm for the quadratic assignment problem. Technical report, DTIC Document, Center for Naval Analyses Arlington VA, 1972.
18 T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proc. 25th Int'l Conf. on Supercomputing*, pages 75–84. ACM, 2011.
19 G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
20 G. Mercier and J. Clet-Ortega. Towards an efficient process placement policy for MPI applications in multicore environments. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroMPI)*, pages 104–115. Springer, 2009.
21 G. Mercier and Emmanuel J. Improving MPI applications performance on multicore clusters with rank reordering. In *18th European MPI Users' Group Meeting*, pages 39–49, 2011.
22 MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.1.
23 H. Müller-Merbach. *Optimale Reihenfolgen*, volume 15 of *Ökonometrie und Unternehmensforschung*. Springer, 1970.
24 P. M. Pardalos and H. Wolkowicz, editors. *Quadratic Assignment and Related Problems, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, May 20-21, 1993*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. DIMACS/AMS, 1994. URL: http://dimacs.rutgers.edu/Volumes/Vol16.html.
25 F. Pellegrini. Scotch Home Page. http://www.labri.fr/pelegrin/scotch.
26 S. Sahni and T. F. Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, 1976. URL: http://doi.acm.org/10.1145/321958.321975, doi:10.1145/321958.321975.
27 P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *12th Int'l Sym. on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*. Springer, 2013.

28 K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.

29 C. Schulz and J. L. Träff. VieM v1.00 – Vienna Mapping and Sparse Quadratic Assignment User Guide. *CoRR*, abs/1703.05509, `http://viem.taa.univie.ac.at/`, 2017. URL: `http://arxiv.org/abs/1703.05509`.

30 A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Global Optimization*, 29(2):225–241, 2004.

31 R. V. Southwell. Stress-Calculation in Frameworks by the Method of "Systematic Relaxation of Constraints". *Proc. of the Royal Society of London*, 151(872):56–95, 1935.

32 J. L. Träff. Implementing the MPI process topology mechanism. In *ACM/IEEE Supercomputing*, 2002.

33 J. T. Vogelstein, J. M. Conroy, V. Lyzinski, L. J. Podrazik, S. G. Kratzer, E. T. Harley, D. E. Fishkind, R. J. Vogelstein, and C. E. Priebe. Fast approximate quadratic programming for graph matching, April 2015. `doi:10.1371/journal.pone.0121002`.

34 C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

35 H. Yu, I-H. Chung, and J. E. Moreira. Topology mapping for Blue Gene/L supercomputer. In *ACM/IEEE Supercomputing*, page 116, 2006.

A Benchmark Instance Properties

Table 3 Benchmark instance properties.

Graph	n	m
UF Graphs		
cop20k_A	99 843	1 262 244
2cubes_sphere	101 492	772 886
thermomech_TC	102 158	304 700
cfd2	123 440	1 482 229
boneS01	127 224	3 293 964
Dubcova3	146 689	1 744 980
bmwcra_1	148 770	5 247 616
G2_circuit	150 102	288 286
shipsec5	179 860	4 966 618
cont-300	180 895	448 799
Large Walshaw Graphs		
598a	110 971	741 934
fe_ocean	143 437	409 593
144	144 649	1 074 393
wave	156 317	1 059 331
m14b	214 765	1 679 018
auto	448 695	3 314 611
Large Other Graphs		
del23	≈8.4M	≈25.2M
del24	≈16.7M	≈50.3M
rgg23	≈8.4M	≈63.5M
rgg24	≈16.7M	≈132.6M
deu	≈4.4M	≈5.5M
eur	≈18.0M	≈22.2M
af_shell9	≈504K	≈8.5M
thermal2	≈1.2M	≈3.7M
nlr	≈4.2M	≈12.5M

The Isomap Algorithm in Distance Geometry

Leo Liberti[1] and Claudia D'Ambrosio[2]

1 **CNRS LIX Ecole Polytechnique, Palaiseau, France**
`liberti@lix.polytechnique.fr`
2 **CNRS LIX Ecole Polytechnique, Palaiseau, France**
`dambrosio@lix.polytechnique.fr`

Abstract

The fundamental problem of distance geometry consists in finding a realization of a given weighted graph in a Euclidean space of given dimension, in such a way that vertices are realized as points and edges as straight segments having the same lengths as their given weights. This problem arises in structural proteomics, wireless sensor networks, and clock synchronization protocols to name a few applications. The well-known Isomap method is a dimensionality reduction heuristic which projects finite but high dimensional metric spaces into the "most significant" lower dimensional ones, where significance is measured by the magnitude of the corresponding eigenvalues. We start from a simple observation, namely that Isomap can also be used to provide approximate realizations of weighted graphs very efficiently, and then derive and benchmark six new heuristics.

1998 ACM Subject Classification G.1.6 Optimization, G.2.2 Graph Theory, F.2.1 Numerical Algorithms and Problems, J.3 Life and Medical Sciences

Keywords and phrases distance geometry problem, protein conformation, heuristics

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.5

1 Introduction

The fundamental problem in Distance Geometry (DG) is as follows.

Distance Geometry Problem (DGP). Given an integer $K \geq 1$ and a simple, edge-weighted, undirected graph $G = (V, E, d)$, where $d : E \to \mathbb{R}_+$, determine whether there exists *realization function* $x : V \to \mathbb{R}^K$ such that:

$$\forall \{i, j\} \in E \quad \|x_i - x_j\| = d_{ij}. \tag{1}$$

The DGP arises in many applications, for various values of K. Two important applications for $K = 3$ are the determination of protein structure from distance data [36], and the localization of a fleet of unmanned submarine vehicles [2]. The localization of mobile sensors in a wireless network is a well-studied application for the case $K = 2$ [11, 5, 16, 9]. The only engineering application we are aware of for the case $K = 1$ is to clock synchronization protocols in computer networks [32]. Although Equation (1) is actually a schema (since the norm is unspecified), most of the literature about the DGP uses the Euclidean norm (or 2-norm) [21, 19], which is also the focus of this paper. In this context, the name of the problem is Euclidean DGP (EDGP).

It is worth mentioning that, although the system of equations in Equation (1) involves square roots, the squared system

$$\forall \{i, j\} \in E \quad \|x_i - x_j\|^2 = d_{ij}^2. \tag{2}$$

© Leo Liberti and Claudia D'Ambrosio;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 5; pp. 5:1–5:13

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

has the same set of solutions as Equation (1) and is a polynomial system of degree two [12]. This makes it amenable to be studied using methods of algebraic geometry, for example [35]. It was shown in [29] that the EDGP is **NP**-hard, by reduction from Partition to the case $K = 1$. Another proof for $K = 2$ was sketched in [38], and further proofs for some variants in $K = 3$ and general K were given in [17] and [20]. If the dimensionality K is not given in advance, then the question is whether the given graph admits a realization in some dimension K. This problem is known as Euclidean Distance Matrix Completion Problem (EDMCP). This difference between being given a K as part of the input or obtaining K as part of the output is considerable: while the EDGP is **NP**-hard, we do not know whether the EDMCP is **NP**-hard or in **P** (or neither, assuming $\mathbf{P} \neq \mathbf{NP}$).

Isomap [34] is a well-known dimension reduction algorithm which is able to project a set $X \subset \mathbb{R}^n$ of high dimensional points belonging to a low-dimensional manifold to its intrinsic dimension (say, K).

In this paper, we describe an easy adaptation of the Isomap algorithm to solve the EDGP. The rationale for using Isomap on the EDGP is that finding realizations in high dimensional spaces is empirically easier than in a given dimension K. We then describe six heuristics for the EDGP based on Isomap, and evaluate them computationally on a test set consisting of protein instances of different sizes.

It is often remarked that the EDGP and EDMCP only serve as abstract models for real-life applications, since in most engineering and biological settings only interval estimates or distributions of the distances are known (rather than exact distance values). Although we do not treat the case of intervals or distributions here, we note that it is at least theoretically possible to extend our heuristic methods to the interval case without excessive trouble — the simplest way to do so is to run the same heuristics using the distribution average on each interval. A better approach would replace error measures based on exact distances by corresponding measures in intervals [21].

The rest of this paper is organized as follows. In Section 2, we give a very brief account of the history of DG, introducing some of the concepts which we shall use later on in the paper. In Section 3 we describe the Isomap algorithm and its relationship to EDMCP and EDGP. In Section 4 we define and motivate our new heuristics based on Isomap. Our computational results are discussed in Section 5.

2 A very short history of DG

DG was formally introduced by Karl Menger in [24, 25], at a time when, under Hilbert's drive [14], the concerted effort of many mathematicians (specially from *Mitteleuropa*) pushed towards the axiomatization of mathematics in general, and specifically of geometry [18]. Menger and some of his students (Gödel among them) were part of the Vienna Circle, but when this became politicized, Menger founded his famous *mathematisches Kolloquium*, which ran at the University of Vienna between 1928 and 1937. It is interesting that the only co-authored paper published by Gödel appears in the proceedings of Menger's *Kolloquium*, and is about DG [18, 22]. Menger's foundational work is an axiomatization of geometry which puts metric spaces at its core (e.g. convexity can be defined via *betweenness* of points). Its main achievement is to characterize the metric spaces according to the dimension of the Euclidean spaces they can be realized in. Menger's work was continued by his student Blumenthal [7], but remained firmly in the domain of pure mathematics.[1]

[1] Another useful application dating from ancient Greece was Heron's formula for computing the area of a triangle from the lengths of its sides, extensively used in agricultural measurements.

A finite metric space (V, d) is a finite set V with an associated metric d. It is usually represented as a weighted complete graph or a distance matrix. The graph is simple, undirected and edge-weighted, say $G = (V, E, d)$ where $E = \{\{i, j\} \mid i < j \in V\}$ and $d : E \to \mathbb{R}_+$ such that $d(i, j)$ is the value of the metric on the edge $\{i, j\}$ of the underlying set V. The distance matrix is an $n \times n$ symmetric matrix D with zero diagonal, where $n = |V|$, such that the component d_{ij} is the value of the metric defined on i and j, for all $i < j \in V$. Given some positive integer K, a metric space (V, d) is *realized* in the Euclidean space $\mathbb{R}^K$ w.r.t. $\|\cdot\|$ if there exists a realization function from V to $\mathbb{R}^K$ w.r.t. $\|\cdot\|$. The main problem in DG, for Menger and Blumenthal, was that of categorizing finite metric spaces (V, d) according to the integers K such that V can be realized in $\mathbb{R}^K$.

A note [31] written by Schoenberg's in 1935 on a paper by Fréchet showed that any Euclidean Distance Matrix (EDM) $D = (d_{ij})$, i.e. when the metric is the 2-norm, can be efficiently transformed into the Gram matrix of a Euclidean realization of the underlying metric. Since a matrix X is Gram if and only if it is Positive Semidefinite (PSD), and since PSD matrices can be factored as $X = xx^\top$ where x is a matrix of rank $K \leq n$, this offers a method for finding a realization of D in $\mathbb{R}^K$ [33]. This result was subsequently adapted to work on wrong or approximate EDMs by replacing negative eigenvalues of D by zero, and resulted in the hugely successful *multidimensional scaling* (MDS) method [8]. A further refinement, obtained by using only at most K positive eigenvalues of D, called *principal component analysis* (PCA) was equally successful. This firmly establishes DG as a branch not only of pure, but also of applied mathematics.

The first explicit mention of the DGP appears to arise in a 1978 paper by Yemini [37], which calls the reader's attention to the problem of finding a realization in the plane of a set of mobile sensors where the distances are only known if two sensors are close enough.

3 The Isomap method in Distance Geometry

The Isomap algorithm projects a finite subset of points $X \subset \mathbb{R}^n$ to $\mathbb{R}^K$ (for some positive given $K < n$) as follows:

1. it computes all pairwise distances for X, yielding the distance matrix D
2. it selects a subset d of "short" Euclidean distances in D (usually up to a given threshold), yielding a simple connected weighted graph $G = (V, E, d)$ where $d : E \to \mathbb{R}_+$;
3. it computes all shortest paths in G, and produces an approximate distance matrix $\tilde{D}$, where $\tilde{D}_{ij} = d_{ij}$ for all $\{i, j\} \in E$ and $\tilde{D}_{ij}$ is the value of the shortest path from i to j otherwise;
4. it derives a corresponding approximate Gram matrix $\tilde{B}$ by setting
$$\tilde{B} = -\frac{1}{2} J \tilde{D}^2 J, \tag{3}$$
where $J = I_n - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$;
5. it finds the (diagonal) eigenvalue matrix Λ of $\tilde{B}$ and the corresponding eigenvector matrix P, so that $\tilde{B} = P^\top \Lambda P$;
6. since $\tilde{B}$ is only an approximation of a Gram matrix, it might have some negative eigenvalues: Isomap replaces all the negative eigenvalues with zeroes;
7. in case there are still more than K positive eigenvalues, Isomap replaces the smallest ones, leaving only the largest K eigenvalues on the diagonal of a PSD matrix $\tilde{\Lambda}$;
8. finally, it sets $x = P^\top \sqrt{\tilde{\Lambda}}$.

Steps 4–8 are collectively known as PCA. Without Step 7, they are known as classic MDS [15].

3.1 Isomap and the EDMCP

How can Isomap apply to the EDGP? A simple explanation is as follows: solving the EDGP is hard, but solving the EDMCP is not as hard, and provides a realization x' of G in (generally) more than K dimensions, say in $\mathbb{R}^n$. At this point, Isomap could be applied to x' and give an approximate projection in $\mathbb{R}^K$.

Although it was mentioned in Section 1 that no-one knows yet whether the EDMCP is **NP**-hard or in **P**, that statement refers to the usual definition of these complexity classes in the the Turing Machine (TM) computational model. On the other hand, the fact that the EDMCP can be solved efficiently in practice can be made more precise.

▶ **Theorem 1.** *The EDMCP can be solved in a polynomial number of basic steps in the Real RAM computational model [6].*

Proof. We first show that the EDMCP can be described by the following pure feasibility Semidefinite Program (SDP):

$$\left.\begin{array}{rrcl} \forall \{i,j\} \in E & X_{ii} + X_{jj} - 2X_{ij} & = & d_{ij}^2 \\ & X & \succeq & 0. \end{array}\right\} \tag{4}$$

Assume Equation (4) has a solution X^* for a given EDMCP instance. Then, since X^* is a PSD matrix, it is also a Gram matrix, which means that it can be factored as $X^* = YY^\top$. Consider a realization $x^* \in \mathbb{R}^n$ given by $x_i^* = Y_i$ for each $i \in V$, where Y_i is the i-th row of Y. Then we have

$$\|x_i^* - x_j^*\|^2 = \|Y_i - Y_j\|^2 = Y_i^\top Y_i + Y_j{}^\top Y_j - 2Y_iY_j = X_{ii} + X_{jj} - 2X_{ij} = d_{ij}^2$$

by the linear constraints in Equation (4). This means that Y is a valid realization for G in $\mathbb{R}^n$, i.e. the given EDMCP instance is YES. Assume now that Equation (4) is infeasible, but suppose that the given instance is YES: then it has a realization Y of G in $\mathbb{R}^n$, and it is immediate to verify that its Gram matrix $X^* = YY^\top$ satisfies Equation (4) providing a contradiction, so the given EDMCP instance must be NO, which concludes the first part of the proof.

Having established that solving the EDMCP is the same as solving Equation (4), we remark that the Interior Point Method (IPM) can be used to solve SDPs in polynomial time to any desired accuracy [1] in the TM computational model. If a primal path-following IPM based on Newton's steps could be run on a Real RAM machine, it would find an exact real solution for the EDMCP. ◀

In practice, the IPM can only compute approximate solutions to Equation (4) in floating point precision, which might not satisfy the constraints exactly. But SDP technology can be used in practice to solve EDMCPs efficiently to very good approximations.

3.2 Isomap and the EDGP

Although our basic idea is to solve an EDMCP instance in order to find a high dimensional realization of G as a pre-processing step to applying Isomap, it is easy to streamline this procedure better. We observe that Step 3 requires a weighted graph G as input, and that a weighted graph is part of the definition of any EDGP instance. It is therefore sufficient to start Isomap from Step 3. The *Isomap for DG* works as follows.

(A) Run the Floyd-Warshall all-shortest-paths algorithm [23] on the partial distance matrix D^G represented by $G = (V, E, d)$ and obtain $\tilde{D}$, a *completion* of D^G;

(B) find the (approximate) Gram matrix $\tilde{B}$ of $\tilde{D}$;
(C) find the PSD matrix B' closest to $\bar{B}$ by zeroing the negative eigenvalues, and then perform PCA to extract an approximate realization x in $\mathbb{R}^K$.

The interest in using the Isomap for DG is that it lends itself to the construction of many heuristics, through its combination with various pre- and post-processing algorithms. For example, Step 1, which essentially aims at solving an EDMCP instance by completing the corresponding graph using shortest paths, can be replaced by the solution of the SDP in Equation (4). Moreover, the final solution x obtained in Step 3 can be used as a starting point by a local Nonlinear Programming (NLP) solver.

4 Isomap heuristics

We list in this section six new heuristics based on Isomap for solving EDGPs.

(i) Isomap. This is the Isomap algorithm for DG as described in Section 3.2.
(ii) IsoNLP. This variant adds a post-processing phase consisting of a local NLP solver to improve the output of Isomap (see Section 4.1 below). Because of the importance of this phase, every following heuristic also uses it.
(iii) SPT. This variant, the name of which stands for *spanning tree*, replaces Step 1 of the Isomap algorithm definition given in Section 3.2 as follows: compute a realization $x' \in \mathbb{R}^K$ using a spanning tree of G (see Section 4.2 below for details). This realization is then used to obtain the EDM $\tilde{D}$. SPT also adds a post-processing local NLP solution phase.
(iv) SDP. This variant replaces Step 1: solve Equation (4) using a "natural" SDP formulation (see Section 4.3 below), obtain a realization x' in $\mathbb{R}^n$, and use it to compute the EDM $\tilde{D}$. SDP also adds a post-processing local NLP solution phase.
(v) Barvinok. This variant is similar to SDP, but it endows the SDP with an objective function designed to decrease the rank of the solution x' to $p = O(\sqrt{|E|})$ (see Section 4.4 below). Barvinok also adds a post-processing local NLP solution phase.
(vi) DGSol. This variant uses one of the first modern algorithms for solving EDGPs, `dgsol` [26], to compute an initial realization x' in $\mathbb{R}^K$, which it then uses to compute the EDM $\tilde{D}$ (see Section 4.5 below). DGSol also adds a post-processing local NLP solution phase.

4.1 Post-processing using a local NLP solver (IsoNLP)

Although Isomap can be used on its own, the quality of the realizations it obtains is greatly improved when the output is used as a starting point for a local NLP algorithm, such as active set or barrier algorithms [13]. Aside from Isomap, the rest of our heuristics all include this post-processing phase.

In the case of the SDP and Barvinok heuristics, this post-processing is backed by a theoretical result given in [4], also exploited in [10], which states that there is an SDP solution of Equation (4) which is asymptotically not too far from the manifold of solutions of Equation (2): hence, it makes sense to try a single local descent to reach that manifold.

The choice of solver was carried out through some preliminary computational experiments. Since most tests were carried out in Python, we limited ourselves to solvers offering a Python API. Among these, we decided to use the one which proved out to be *empirically* fastest, namely `lbfgs` from `scipy.optimize`, limited to 50 iterations. Code optimization might change this choice, specially in view of the fact that we only employed Python-enabled solvers, with APIs that shine more for ease of coding than efficiency.

4.2 Spanning tree realization heuristic (SPT)

A possible way to construct an approximate distance matrix $\tilde{D}$ based on an EDGP instance G consists in identifying a spanning subgraph of G for which the EDGP can be solved efficiently, and then use the corresponding realization to compute $\tilde{D}$. We use trees, which are a polynomial case of the EDGP.

Let T be a tree on V, and for each $v \in V$ let $N_T(v)$ be the set of vertices adjacent to v in T. The following algorithm realizes any tree in $K = 1$, and more specifically in the non-negative half-line $\mathbb{R}_+$.

1. Let r be a vertex with highest degree in G;
2. let $x_r = 0$;
3. let $Q = \{r\}$ be a priority queue containing vertices with their degrees w.r.t. $V \smallsetminus Q$ as priority;
4. pop the vertex u with highest priority from Q;
5. for each $v \in N_T(u)$ let $x_v = x_u + d_{uv}$ and add v to Q.

▶ **Lemma 2.** *The above algorithm is correct and runs in linear time.*

Proof. The important invariant of the algorithm is that every vertex u entering Q has a known realization x_u: this holds by Step 5 and because at the first iteration Q only contains r, realized at $x_r = 0$. It is also easy to see that, by connectedness of G, every vertex in V enters Q at least once. Moreover, since G is a tree, it has no cycles, which implies that no vertex can ever enter Q more than once. Since every vertex enters Q exactly once, the complexity of this algorithm is $\Theta(|V|)$. ◀

Once a tree is realized in a half-line, one can embed the realization in as many dimensions as needed, by embedding a congruent copy of the half-line in an appropriate Euclidean space.

The algorithm we use to construct a realization in $\mathbb{R}^K$ from a tree is based on the above one. It takes a general graph G as input, grows a largest-degree priority spanning tree, and realizes each vertex v as a uniformly chosen random point on the sphere centered at x_u with radius d_{uv} for each edge $\{u, v\}$ in the spanning tree. More precisely, we modify the above algorithm as follows: (a) we introduce a set Z (initialized to $\{r\}$) that records the vertices entering the tree and replace $N_T(u)$ by $N_G(u) \smallsetminus Z$; (b) we replace $x_v = x_u + d_{uv}$ by x_v sampled uniformly at random from $S^{K-1}(x_u, d_{uv})$.

4.3 Euclidean distance SDP objective (SDP)

Another way to compute $\tilde{D}$ is to solve the SDP in Equation (4), and obtain a realization Y having rank generally higher than K; $\tilde{D}$ is then set to the EDM corresponding to Y. IPM algorithms for SDP offer us some additional flexibility in that they solve optimization problems rather than pure feasibility problems such as Equation (4).

In the SDP heuristic, we simply rewrite the pure feasibility SDP as an optimization problem. First, we reformulate Equation (2) as follows:

$$\left.\begin{array}{rcl} \min & \sum\limits_{\{i,j\}\in E} \|x_i - x_j\|^2 & \\ \forall \{i,j\} \in E & \|x_i - x_j\|^2 \ \geq & d_{ij}^2. \end{array}\right\} \quad (5)$$

The objective function of Equation (5) "pulls together" the realizations of the vertices, limited to the minimum possible value d_{ij} of each pairwise distance in E because of the

constraints. Notice that Equation (5) has a convex objective but reverse convex constraints, both linearized in the SDP relaxation below [10]:

$$\left.\begin{array}{rrcl} \min\limits_{X\succeq 0} & \sum\limits_{\{i,j\}\in E}(X_{ii}+X_{jj}-2X_{ij}) & & \\ \forall\{i,j\}\in E & X_{ii}+X_{jj}-2X_{ij} & \geq & d_{ij}^2, \end{array}\right\} \quad (6)$$

Once the SDP solver finds a feasible solution X^* for Equation (6), one can either compute $\tilde{D}$ by inverting Equation (3) with $\tilde{B} = X^*$, or factor X^* into $YY^\top$ and then use the realization Y in $\mathbb{R}^n$ to compute $\tilde{D}$ as the corresponding EDM.

4.4 Barvinok's result (Barvinok)

The Barvinok heuristic is very similar to the SDP heuristic, but we solve a different SDP: more precisely, we endow Equation (4) with an objective function $\min F \bullet X$ for some "regular" matrix F (we sketch the regularity definition below). Barvinok proves in [3] that this SDP will generally provide a realization Y having rank $O(\sqrt{|E|})$ rather than $O(n)$. We recall that $F \bullet Y$ is the trace of the dot product of $F^\top$ and X. Write the columns of F one after the other to obtain a column vector in $\mathbb{R}^{n^2}$, and do the same for X: then $F \bullet X$ corresponds to the "ordinary" scalar product between these vectors.

More precisely, Barvinok proves that if a graph is realizable in $\mathbb{R}^t$ for some t which is generally $O(n)$, then it is also realizable for $t = \lfloor(\sqrt{8|E|+1}-1)/2\rfloor$. The way he achieves this result is by showing that there exists a solution X to the SDP

$$\left.\begin{array}{rrcl} \min\limits_{X\succeq 0} & & F\bullet X & \\ \forall\{i,j\}\in E & X_{ii}+X_{jj}-2X_{ij} & = & d_{ij}^2, \end{array}\right\} \quad (7)$$

having rank $\leq t$, and that there exist matrices F which yield these low-rank solutions. Specifically, F must be "regular" with respect to the quadratic forms involved in the constraints of Equation (7). In other words, if we write $X_{ii} + X_{jj} - 2X_{ij}$ as $Q^{ij} \bullet X$ for some real symmetric $n \times n$ matrix Q^{ij} (for each $\{i, j\} \in E$), then F must be "regular" with respect to all the Q^{ij}s.

Regularity, in this context, is defined by two conditions, one easy to explain and one harder. First, and easiest, F should be positive definite (PD), namely all its eigenvalues should be strictly positive. The hard part is as follows: consider the set of $n \times n$ matrices X that are feasible in (4): since each such X is PSD, it can be factored into $YY^\top$. As X ranges over the feasible region of Equation (7), Y ranges over possible realizations of G having various ranks $r \leq n$. We let this range be $\mathcal{Y} = \{Y \in \mathbb{R}^{r\times n} \mid YY^\top = X \text{ satisfies } (4) \wedge 1 \leq r \leq n\}$. Now we partition $\mathcal{Y}$ according to the values of r:

$$\forall r \leq n \quad \mathcal{Y}_r = \{Y \in \mathbb{R}^{n\times r} \mid YY^\top = X \text{ satisfies } (4)\}.$$

We then require that the map $\psi_F : \mathbb{R}^{|E|} \to \mathbb{R}^{n\times n}$ given by $\psi_F(z) = F - \sum\limits_{\{i,j\}\in E} z_{ij}Q^{ij}$ intersects each $\mathcal{Y}_r$ transversally, for each $r \leq n$. A map ϕ between smooth manifolds $A \to B$ *intersects a submanifold $B' \subseteq B$ transversally* if either $\phi(A)$ has no intersection with B' at all or, if it does, the intersection points are "well behaved", meaning their tangent spaces are non-singular in a certain way. Explaining this more precisely would require the introduction of too many new concepts; to give a suggestion, the curves $\psi(z) = z^2 + c$ intersect the manifold $z = 0$ transversally as long $c \neq 0$, since at $c = 0$ the tangent of $\psi(z)$ at the intersection point is parallel to the manifold $z = 0$.

In summary, F is regular if it is positive definite (PD) and the map $\psi_F(z)$ intersects each $\mathcal{Y}_r$ transversally. Since regular matrices prevent a corresponding (linear) map from displaying some type of singularity, most PD matrices are regular once the quadratic forms are fixed. Indeed, Barvinok's paper does not even suggest a way to sample or construct such matrices.

In [3, Example 4.1], Barvinok exploits the special structure of r-diagonal matrices to prove that the rank reduction is improved if the Q^{ij} are r-diagonal. Since our quadratic forms are fixed, and not r-diagonal in general, this is hardly relevant to our case. On the other hand, strictly diagonally dominant r-diagonal matrices are PD, so this suggests a good way to randomly generate regular matrices. We recall that a matrix F is r-diagonal if it consists of a diagonal band of width $2r+1$, i.e. an identity with r (small) nonzero entries to the left and right of the i-th diagonal entry. The off-diagonal nonzeros should be small enough for the matrix to be diagonally dominant and, in particular, PD.

4.5 Moré-Wu's `dgsol` algorithm

The `dgsol` algorithm has an outer iteration and an inner one [26]. The outer iteration starts from a smoothed convexified version of the penalty objective function

$$f(x) = \sum_{\{i,j\}\in E} \left(\|x_i - x_j\|_2^2 - d_{ij}^2\right)^2 \tag{8}$$

obtained via a Gaussian transform

$$\langle f\rangle_\lambda(x) = \frac{1}{\pi^{Kn/2}\lambda^{Kn}} \int\limits_{\mathbb{R}^{Kn}} f(y)\exp(-\|y-x\|_2^2/\lambda^2)\mathsf{d}y, \tag{9}$$

which tends to $f(x)$ as $\lambda \to 0$. For each fixed value of λ in the outer iteration, the inner iteration is based on the step

$$x^{\ell+1} = x^\ell - \alpha_\ell H_\ell \nabla\langle f\rangle_\lambda(x^\ell),$$

for $\ell \in \mathbb{N}$, where α_ℓ is a step size, and H_ℓ is an approximation of the inverse Hessian matrix of f. In other words, the inner iteration implements a local NLP solution method which uses the optimum at the previous value of λ as a starting point.

Overall, this yields a homotopy method which traces a trajectory as a function of $\lambda \to 0$, where a unique (global) optimum of the convex smoothed function $\langle f\rangle_\lambda$ for a high enough value of λ (hopefully) follows the trajectory to the global minimum of the multimodal, nonconvex function $\langle f\rangle_0 = f$.

The initial solution could theoretically be obtained by setting λ large enough so that $\langle f\rangle_\lambda(x)$ is convex, but DGSol's initial solution is computed by means of a spanning tree realization instead (see Section 4.2).

5 Computational assessment

We consider two test sets: a larger test set based on instances of various sizes, and a smaller test set with five very large sized instances. All instances are protein instances derived from Protein Data Bank (PDB) files, which contain realizations in $\mathbb{R}^3$. In order to obtain realistic instances, we computed the EDMs of these realizations, then kept the partial distance matrix consisting of all distances within a threshold of 5Å. This generates instances that are similar to the type of distance data produced by Nuclear Magnetic Resonance (NMR) experiments [30].

All of the heuristics have been coded in Python 2.7. All the tests have been obtained on a single core of an Intel Xeon processor at 2.53GHz with 48GB RAM. SDPs were solved using Mosek 7.1.0.41 [27] through the PICOS [28] Python-based API.

5.1 Small to large sizes

We consider a set of 25 protein instances with sizes ranging from $n = 15$, $|E| = 39$ to $n = 488$, $|E| = 5741$, detailed in Table 1. For each instance and solution method we record the (scaled) mean (MDE) and largest (LDE) distance error of the solution, defined as:

$$\mathsf{mde}(x) = \frac{1}{|E|} \sum_{\{i,j\} \in E} \frac{|\, \|x_i - x_j\|_2 - d_{ij} \,|}{d_{ij}};$$

$$\mathsf{lde}(x) = \frac{1}{|E|} \max_{\{i,j\} \in E} \frac{|\, \|x_i - x_j\|_2 - d_{ij} \,|}{d_{ij}}$$

as a measure of the solution quality, as well as the CPU time taken by each method. All CPU times have been computed in Python using the `time` module. The last three lines of Table 1 contain geometric means, averages and standard deviations. Best results are emphasized in boldface.

According to Table 1, IsoNLP and Barvinok are the best heuristics with respect to both MDE and LDE, whereas Isomap is the fastest (but quality-wise among the worst).

5.2 Very large sizes

For the large sized instance benchmark, we considered a set of five very large protein datasets, detailed in Table 2. None of the heuristics above, aside from Isomap, was able to terminate within 12h of CPU time, the issue being that our post-processing phase based on Python's `scipy.optimize.lbfgs` local NLP solver is too slow. The results obtained by Isomap, however, are quite poor qualitywise, with MDE and LDE measures attaining values around 0.99 for all five instances.

We therefore decided to re-implement[2] IsoNLP using a fully compiled language (a mixture of pure C and Fortran 77). We use the Isomap algorithm to obtain a starting point for `dgopt`, the optimization engine used by `dgsol`. Since `dgopt` is a homotopy method rather than a simple NLP solver, we thought it would be fair to compare this heuristic with `dgsol` itself, which uses a spanning tree heuristic (see Section 4.2) to provide a starting point to `dgopt`. The results of our tests are presented in Table 2.

The results do not show a clear quality-wise dominance of either solution method. CPU time wise, DGSol has a clear advantage: this is easily explained since IsoNLP differs from DGSol only by the choice of the initial (approximate) realization, which takes $O(n^3)$ in IsoNLP (finding eigenvalues and eigenvectors within IsoMAP) and $O(n)$ in DGSol (Lemma 2). Given the large lde values, by our past experience the only reliable solution in Table 2 is the `water` realization obtained by IsoNLP (see Figure 1).

Methodologically, there is a generality vs. efficiency trade-off at play in evaluating IsoNLP versus DGSol. Whereas the former applies to the EDGP for any given K, the latter only solves EDGP instances with K fixed to the constant 3 (this trade-off does not concern our implementation, which calls parts of the `dgsol` program). Speficically, the evaluation of the integral in Equation (9) depends on a $\mathsf{d}y = dy_1 \cdots dy_K$ which explicitly depends on K.

[2] See `http://www.lix.polytechnique.fr/~liberti/isomapheur.zip`.

Table 1 Comparative results on small to large sized protein instances ($K = 3$).

Instance			mde						lde						CPU					
Name	n	$\lvert E\rvert$	Isomap	IsoNLP	SPT	SDP	Barvinok	DGSol	Isomap	IsoNLP	SPT	SDP	Barvinok	DGSol	Isomap	IsoNLP	SPT	SDP	Barvinok	DGSol
C0700odd.1	15	39	0.585	0.001	0.190	0.068	**0.000**	0.135	0.989	0.004	0.896	0.389	**0.001**	0.634	**0.002**	1.456	1.589	0.906	1.305	1.747
C0700odd.2	15	39	0.599	**0.000**	0.187	0.086	**0.000**	0.128	0.985	**0.002**	0.956	0.389	0.009	1.000	**0.003**	1.376	1.226	1.002	1.063	0.887
C0700odd.3	15	39	0.599	**0.000**	0.060	0.086	**0.000**	0.128	0.985	**0.002**	0.326	0.389	0.009	1.000	**0.003**	1.259	1.256	0.861	1.167	0.877
C0700odd.4	15	39	0.599	**0.000**	0.283	0.086	0.001	0.128	0.985	**0.002**	2.449	0.389	0.008	1.000	**0.003**	1.347	1.222	0.976	1.063	1.033
C0700odd.5	15	39	0.599	**0.000**	0.225	0.086	0.000	0.128	0.985	**0.002**	0.867	0.389	0.007	1.000	**0.003**	1.284	1.157	0.987	1.100	0.700
C0700odd.6	15	39	0.599	**0.000**	0.283	0.086	0.000	0.128	0.985	**0.002**	1.520	0.389	**0.002**	1.000	**0.002**	1.372	1.196	0.998	1.305	0.909
C0700odd.7	15	39	0.585	0.001	0.080	0.068	**0.000**	0.135	0.989	0.004	0.361	0.389	**0.001**	0.634	**0.003**	1.469	1.322	0.894	1.093	1.719
C0700odd.8	15	39	0.585	0.001	0.056	0.068	**0.000**	0.135	0.989	0.004	0.275	0.389	**0.003**	0.634	**0.003**	1.408	1.306	0.692	1.079	1.744
C0700odd.9	15	39	0.585	0.001	0.057	0.068	**0.000**	0.135	0.989	0.004	0.301	0.389	**0.002**	0.634	**0.002**	1.430	1.172	0.791	1.093	1.745
C0700odd.A	15	39	0.585	0.001	0.043	0.068	**0.000**	0.135	0.989	**0.004**	0.316	0.389	**0.004**	0.634	**0.002**	1.294	1.269	0.722	1.220	1.523
C0700odd.B	15	39	0.585	0.001	0.151	0.068	**0.000**	0.135	0.989	**0.004**	1.022	0.389	**0.004**	0.634	**0.002**	1.297	1.279	0.871	1.111	1.747
C0700odd.C	15	39	0.835	**0.022**	0.033	0.039	0.031	0.025	1.012	**0.147**	0.393	0.211	0.294	0.167	**0.004**	6.803	6.369	7.371	7.030	7.000
C0700odd.D	36	242	0.835	**0.022**	0.041	0.039	0.042	0.025	1.012	**0.147**	0.423	0.211	0.268	0.167	**0.006**	6.806	6.575	7.422	7.603	7.095
C0700odd.E	36	242	0.835	**0.022**	0.064	0.039	0.031	0.025	1.012	**0.147**	0.894	0.211	0.260	0.167	**0.006**	6.911	6.638	7.365	6.979	7.008
C0700odd.F	36	242	0.599	**0.000**	0.047	0.086	**0.000**	0.128	0.985	**0.002**	0.308	0.389	0.005	1.000	**0.002**	1.299	1.310	1.008	1.100	1.040
C0150alter.1	37	335	0.786	0.058	0.066	0.014	0.015	**0.010**	0.992	0.571	0.693	0.256	0.285	**0.253**	**0.004**	9.492	9.456	10.276	10.120	9.272
C0080create.1	60	681	0.887	0.053	0.083	**0.024**	**0.024**	0.054	1.967	0.949	0.789	**0.511**	0.516	0.718	**0.012**	18.835	19.720	21.247	20.906	19.962
C0080create.2	60	681	0.887	0.053	0.047	**0.024**	**0.024**	0.054	1.967	0.949	0.585	**0.511**	0.512	0.718	**0.008**	18.791	20.009	21.728	20.885	19.740
C0020pdb	107	999	0.939	0.110	0.119	**0.059**	0.060	0.103	1.242	1.113	1.349	1.082	1.138	**0.798**	**0.035**	29.024	27.772	35.273	35.486	32.479
1guu	150	955	0.986	0.068	0.069	**0.057**	**0.057**	0.061	0.999	0.854	0.830	**0.735**	0.751	0.768	**0.048**	30.869	28.784	41.488	41.852	37.848
1guu-1	150	959	0.986	0.061	0.063	0.058	**0.057**	0.060	1.000	**0.711**	0.855	0.805	0.829	0.778	**0.053**	31.322	31.442	42.308	41.590	37.218
1guu-4000	150	968	0.974	0.081	0.080	0.072	**0.065**	0.079	1.000	0.901	**0.728**	0.760	0.961	0.826	**0.050**	30.352	29.856	42.330	39.832	42.015
C0030pkl	198	3247	0.961	0.112	0.160	**0.076**	0.077	0.137	**1.197**	1.354	2.230	1.995	2.054	1.401	**0.091**	105.175	104.775	149.192	146.360	111.859
1PPT	302	3102	0.984	**0.121**	0.129	0.128	0.129	0.123	**1.000**	1.519	1.219	1.944	1.956	1.224	**0.356**	112.448	110.345	185.815	187.182	118.681
100d	488	5741	0.987	0.146	0.146	0.155	0.157	**0.137**	**1.000**	1.577	1.397	1.764	1.749	1.358	**0.828**	229.809	213.136	659.638	659.280	233.115
GeoMean			0.74	**0.00**	0.09	0.06	**0.00**	0.08	1.07	**0.04**	0.73	0.50	0.06	0.66	**0.01**	6.30	6.04	5.93	6.63	6.30
Avg			0.76	0.04	0.11	0.07	**0.03**	0.10	1.09	**0.44**	0.88	0.63	0.47	0.77	**0.06**	26.12	25.21	49.69	49.55	27.96
StDev			0.17	0.05	0.07	**0.03**	0.04	0.04	**0.27**	0.55	0.57	0.52	0.65	0.34	**0.18**	51.69	48.82	135.08	134.97	53.26

Table 2 Tests on large protein instances ($K = 3$).

Instance			mde		lde		CPU	
Name	$\|V\|$	$\|E\|$	IsoNLP	dgsol	IsoNLP	dgsol	IsoNLP	dgsol
water	648	11939	**0.005**	0.15	**0.557**	0.81	26.98	**15.16**
3al1	678	17417	0.036	**0.007**	0.884	**0.810**	**170.91**	210.25
1hpv	1629	18512	**0.074**	0.078	0.936	**0.932**	374.01	**60.28**
il2	2084	45251	**0.012**	0.035	**0.910**	0.932	465.10	**139.77**
1tii	5684	69800	0.078	**0.077**	0.950	**0.897**	7400.48	**454.375**

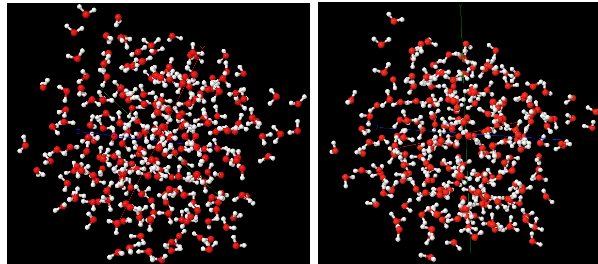


Figure 1 Comparison between water from the PDB (left) and the same structure reconstructed using IsoNLP (right).

6 Conclusion

This paper is concerned with Isomap-based heuristics for solving the Euclidean Distance Geometry Problem. It discusses the Isomap algorithm in the context of Distance Geometry, proposes six new heuristics, and benchmarks them on a set of protein conformation instances of various sizes.

References

1. F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5(1):13–51, 1995.
2. A. Bahr, J. Leonard, and M. Fallon. Cooperative localization for autonomous underwater vehicles. *International Journal of Robotics Research*, 28(6):714–728, 2009.
3. A. Barvinok. Problems of distance geometry and convex properties of quadratic maps. *Discrete and Computational Geometry*, 13:189–202, 1995.
4. A. Barvinok. Measure concentration in optimization. *Mathematical Programming*, 79:33–53, 1997.
5. P. Biswas, T. Lian, T. Wang, and Y. Ye. Semidefinite programming based algorithms for sensor network localization. *ACM Transactions in Sensor Networks*, 2:188–220, 2006.
6. L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: *NP*-completeness, recursive functions, and universal machines. *Bulletin of the American Mathematical Society*, 21(1):1–46, 1989.
7. L. Blumenthal. *Theory and Applications of Distance Geometry*. Oxford University Press, Oxford, 1953.
8. T. Cox and M. Cox. *Multidimensional Scaling*. Chapman & Hall, Boca Raton, 2001.
9. M. Cucuringu, Y. Lipman, and A. Singer. Sensor network localization by eigenvector synchronization over the Euclidean group. *ACM Transactions on Sensor Networks*, 8:1–42, 2012.

10 G. Dias and L. Liberti. Diagonally dominant programming in distance geometry. In R. Cerulli, S. Fujishige, and R. Mahjoub, editors, *International Symposium in Combinatorial Optimization*, volume 9849 of *LNCS*, pages 225–236, New York, 2016. Springer.

11 L. Doherty, K. Pister, and L. El Ghaoui. Convex position estimation in wireless sensor networks. In *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3 of *INFOCOM*, pages 1655–1663, Piscataway, 2001. IEEE.

12 I. Dokmanić, R. Parhizkar, J. Ranieri, and M. Vetterli. Euclidean distance matrices: Essential theory, algorithms and applications. *IEEE Signal Processing Magazine*, 1053-5888:12–30, Nov. 2015.

13 R. Fletcher. *Practical Methods of Optimization.* Wiley, Chichester, second edition, 1991.

14 D. Hilbert. *Grundlagen der Geometrie.* Teubner, Leipzig, 1903.

15 I. Jolliffe. *Principal Component Analysis.* Springer, Berlin, 2nd edition, 2010.

16 N. Krislock and H. Wolkowicz. Explicit sensor network localization using semidefinite representations and facial reductions. *SIAM Journal on Optimization*, 20:2679–2708, 2010.

17 C. Lavor, L. Liberti, N. Maculan, and A. Mucherino. The discretizable molecular distance geometry problem. *Computational Optimization and Applications*, 52:115–146, 2012.

18 L. Liberti and C. Lavor. Six mathematical gems in the history of distance geometry. *International Transactions in Operational Research*, 23:897–920, 2016.

19 L. Liberti, C. Lavor, N. Maculan, and A. Mucherino. Euclidean distance geometry and applications. *SIAM Review*, 56(1):3–69, 2014.

20 L. Liberti, C. Lavor, and A. Mucherino. The discretizable molecular distance geometry problem seems easier on proteins. In A. Mucherino, C. Lavor, L. Liberti, and N. Maculan, editors, *Distance Geometry: Theory, Methods, and Applications*, pages 47–60. Springer, New York, 2013.

21 L. Liberti, C. Lavor, A. Mucherino, and N. Maculan. Molecular distance geometry methods: from continuous to discrete. *International Transactions in Operational Research*, 18:33–51, 2010.

22 L. Liberti, G. Swirszcz, and C. Lavor. Distance geometry on the sphere. In *JCDCG*^{2}, LNCS, New York, accepted. Springer.

23 K. Mehlhorn and P. Sanders. *Algorithms and Data Structures.* Springer, Berlin, 2008.

24 K. Menger. Untersuchungen über allgemeine Metrik. *Mathematische Annalen*, 100:75–163, 1928.

25 K. Menger. New foundation of Euclidean geometry. *American Journal of Mathematics*, 53(4):721–745, 1931.

26 J. Moré and Z. Wu. Global continuation for distance geometry problems. *SIAM Journal of Optimization*, 7(3):814–846, 1997.

27 Mosek ApS. *The `mosek` manual, Version 7 (Revision 114)*, 2014. (`www.mosek.com`).

28 G. Sagnol. *PICOS: A Python Interface for Conic Optimization Solvers.* Zuse Institut Berlin, 2016. URL: `picos.zib.de`.

29 J. Saxe. Embeddability of weighted graphs in k-space is strongly **NP**-hard. *Proceedings of 17th Allerton Conference in Communications, Control and Computing*, pages 480–489, 1979.

30 T. Schlick. *Molecular modelling and simulation: an interdisciplinary guide.* Springer, New York, 2002.

31 I. Schoenberg. Remarks to Maurice Fréchet's article "Sur la définition axiomatique d'une classe d'espaces distanciés vectoriellement applicable sur l'espace de Hilbert". *Annals of Mathematics*, 36(3):724–732, 1935.

32 A. Singer. Angular synchronization by eigenvectors and semidefinite programming. *Applied and Computational Harmonic Analysis*, 30:20–36, 2011.

33 M. Sippl and H. Scheraga. Cayley-Menger coordinates. *Proceedings of the National Academy of Sciences*, 83:2283–2287, 1986.

34 J. Tenenbaum, V. de Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2322, 2000.

35 L. Wang, R. Mettu, and B. R. Donald. An algebraic geometry approach to protein structure determination from NMR data. In *Proceedings of the Computational Systems Bioinformatics Conference*, Piscataway, 2005. IEEE.

36 K. Wüthrich. Protein structure determination in solution by nuclear magnetic resonance spectroscopy. *Science*, 243:45–50, 1989.

37 Y. Yemini. The positioning problem – a draft of an intermediate summary. In *Proceedings of the Conference on Distributed Sensor Networks*, pages 137–145, Pittsburgh, 1978. Carnegie-Mellon University.

38 Y. Yemini. Some theoretical aspects of position-location problems. In *Proceedings of the 20th Annual Symposium on the Foundations of Computer Science*, pages 1–8, Piscataway, 1979. IEEE. doi:10.1109/SFCS.1979.39.

Distributed Domain Propagation*

Robert Lion Gottwald[1], Stephen J. Maher[2], and Yuji Shinano[3]

1 **Zuse Institute Berlin, Berlin, Germany**
`robert.gottwald@zib.de`
2 **Zuse Institute Berlin, Berlin, Germany**
`maher@zib.de`
3 **Zuse Institute Berlin, Berlin, Germany**
`shinano@zib.de`

Abstract

Portfolio parallelization is an approach that runs several solver instances in parallel and terminates when one of them succeeds in solving the problem. Despite its simplicity, portfolio parallelization has been shown to perform well for modern mixed-integer programming (MIP) and boolean satisfiability problem (SAT) solvers. Domain propagation has also been shown to be a simple technique in modern MIP and SAT solvers that effectively finds additional domain reductions after the domain of a variable has been reduced. In this paper we introduce distributed domain propagation, a technique that shares bound tightenings across solvers to trigger further domain propagations. We investigate its impact in modern MIP solvers that employ portfolio parallelization. Computational experiments were conducted for two implementations of this parallelization approach. While both share global variable bounds and solutions, they communicate differently. In one implementation the communication is performed only at designated points in the solving process and in the other it is performed completely asynchronously. Computational experiments show a positive performance impact of communicating global variable bounds and provide valuable insights in communication strategies for parallel solvers.

1998 ACM Subject Classification G.1.6 Optimization, D.1.3 Concurrent Programming

Keywords and phrases mixed integer programming, parallelization, domain propagation, portfolio solvers

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.6

1 Introduction

A MIP is a problem with the general form:

$$\min\{c^\top x : Ax \leq b, l \leq x \leq u, x_j \in \mathbb{Z}, \text{ for all } j \in I\},$$

with matrix $A \in \mathbb{R}^{m\times n}$, vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $l, u \in (\mathbb{R} \cup \{-\infty, +\infty\})^n$, as well as a set $I \subseteq \{1, \ldots, n\}$, which identifies the subset of variables that are integer. This paper deals with algorithmic approaches that aim to reduce the domain of a variable—methods to increase or decrease components of l or u respectively—within a parallel solver. An algorithmic approach of particular interest is domain propagation.

MIP and SAT solvers employ domain propagation after the domain of a variable has been reduced to find further reductions for variables occurring in the same constraints or clauses.

* This work has been supported by the Research Campus MODAL *Mathematical Optimization and Data Analysis Laboratories* funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM). All responsibility for the content of this publication is assumed by the authors.

© Robert Lion Gottwald, Stephen J. Maher, and Yuji Shinano;
licensed under Creative Commons License CC-BY

In modern branch-and-bound based MIP solvers domain propagation has a major positive impact on performance [2, 3, 4, 17]. It is usually performed at every node of the branch-and-bound tree to exploit the possible additional domain reductions that result from applying branching decisions. Regularly performing domain propagation is advantageous since it is able to achieve domain reductions and detect infeasible nodes with a small computational effort compared to solving the respective linear programming (LP) relaxation.

Beyond the traditional application, domain propagation has been incorporated into many different parts of a MIP solver. Gamrath [11] applied domain propagation during strong branching. This use of domain propagation has been shown to significantly improve the solver performance and reduce the branch-and-bound tree size. The average number of LP iterations for strong branching decreased and better dual bounds were obtained while no more time was spent in strong branching.

Modern solvers only propagate constraints if there is the potential of finding domain reductions; i.e. for general linear constraints at least one variable must have a tighter bound than in the last propagation. In branch-and-bound based solvers this generally occurs after each branching decision. However, there are other reasons why a the domain of a variable might be reduced. For instance, some MIP solvers employ a technique called *reduced cost strengthening* [19] that exploits dual information. Particularly, if a variable has non-zero reduced cost in the LP relaxation of a node, a bound can be inferred given the objective value $\hat{z}$ of a feasible primal solution. Because the variable has non-zero reduced cost, its LP solution value must be at the variable's bound. Furthermore, the reduced cost of this variable tells us how much the objective function changes if the variable moves away from its bound. Thereby a bound can be obtained for the variable, that must be satisfied by any solution with objective value $\hat{z}$ or better. If this technique is employed by using the LP relaxation at the root node, the obtained bound is globally valid.

In MIP solvers parallelization can be employed in a variety of ways [21]. A common approach is to parallelize the branch-and-bound algorithm by processing the subproblems concurrently. Another method is *portfolio parallelization*, sometimes also called *(parallel) racing*. In this form of parallelization multiple solvers with different configurations solve the same problem instance in parallel. The approach can be extended by the communication of global information such as feasible solutions and cutting planes. An efficient implementation of these approaches can be difficult due to the complexity that arises from the synchronization of global information.

Although portfolio parallelization does not distribute the work required to solve an instance, it has been shown to be competitive with the parallelization of the branch-and-bound tree search for a small number of processors [7, 14, 10, 5]. One reason for this is a phenomenon called *performance variability* [15]. It refers to the large differences in the performance of a solver that are observed after alterations expected to entail a neutral performance impact; e.g. setting a different random seed or permuting the problem instance.

2 Parallelization in SCIP

In SCIP [12], one of the fastest non-commercial MIP solvers, different forms of parallelization have been implemented. A deterministic shared memory portfolio parallelization of SCIP, referred to as concurrent SCIP, will be presented. Also, there exists a shared memory parallelization of SCIP called FiberSCIP [25], and a distributed memory parallelization called ParaSCIP [24]. The latter two only differ in the framework used for communication and both aim at parallelizing the tree search, but can also be configured to perform racing

only. This paper compares both shared memory parallelizations, concurrent SCIP, and FiberSCIP.

The main difference between concurrent SCIP and FiberSCIP is the method of communication and the timing for sending and receiving information. In FiberSCIP all communications between solver threads are done via a controller thread, the LoadCoordinator, fully asynchronously. In concurrent SCIP the solvers instead gather the information and then communicate when they reach certain points in the solving process using a shared data structure.

2.1 Concurrent SCIP

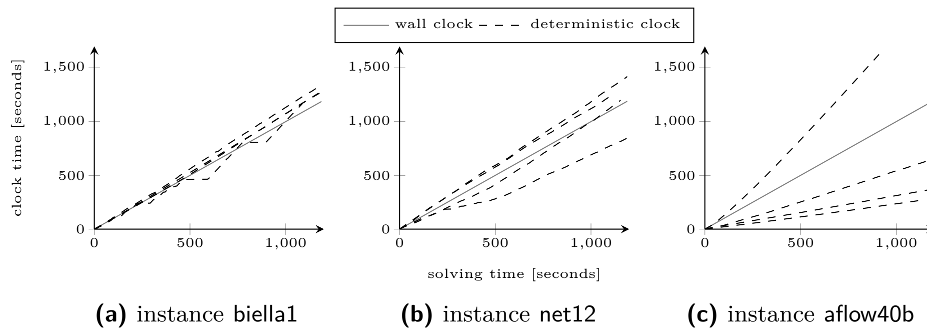
The development of concurrent SCIP was motivated by an effort to exploit performance variability and aid the fast discovery of feasible solutions. The experiments and parts of the description of concurrent SCIP in this section are also contained in the first author's master thesis [13], while the distributed domain propagation is a new feature presented in this paper.

Concurrent SCIP allows to run multiple solver instances in parallel on separate threads. For the purpose of diversification the solvers can be configured to use different settings and random seeds. Additionally, they can share feasible solutions and global variable bounds throughout the solving process. Of particular importance is the sharing of global variable bounds, which is the focus of this paper. The frequency of communication is adjusted dynamically based on the amount of the gap—difference between upper and lower bounds—that was closed between communication points.

Each type of solver used in concurrent SCIP is implemented as a new plugin type of SCIP. Therefore, in addition to SCIP solvers with different parameter settings, other algorithms and solvers could be included into a parallel portfolio. Concurrent SCIP can be compiled either with tinycthread [1], a thin wrapper around the platform specific threads, or with OpenMP [8]. In this paper the tinycthread version is used to compare with FiberSCIP, because they both use Pthreads on Linux.

An important requirement for concurrent SCIP is a deterministic solving process, so that the behavior of the solver can be reproduced. To satisfy this requirement care must be taken when implementing communication between the solvers. In a single-threaded program the sequence of instructions is the same between multiple runs; or at least it appears to be from the programmer's viewpoint even though modern microprocessors reorder instructions internally. In contrast, there are usually millions of different interleavings of instructions that can occur for a single multi-threaded program. One execution of such a program depends on the scheduling decisions of the operating system and other factors that are beyond the control of the programmer. Not only does this make it hard to develop correct multi-threaded applications, it also makes such applications non-deterministic by default. Therefore, in concurrent SCIP the solvers only share information at *communication points*, which are determined by using a deterministic clock [4]. However, a solver must wait if it wants to read information from a communication point that has not yet been reached by all solvers, otherwise it would incur non-determinism. For this reason a deterministic communication scheme can suffer from high idle times. To measure the impact of such behavior, concurrent SCIP can also be configured to use the wall clock instead of the deterministic clock for determining the communication points.

[1] http://tinycthread.github.io/



(a) instance biella1 **(b)** instance net12 **(c)** instance aflow40b

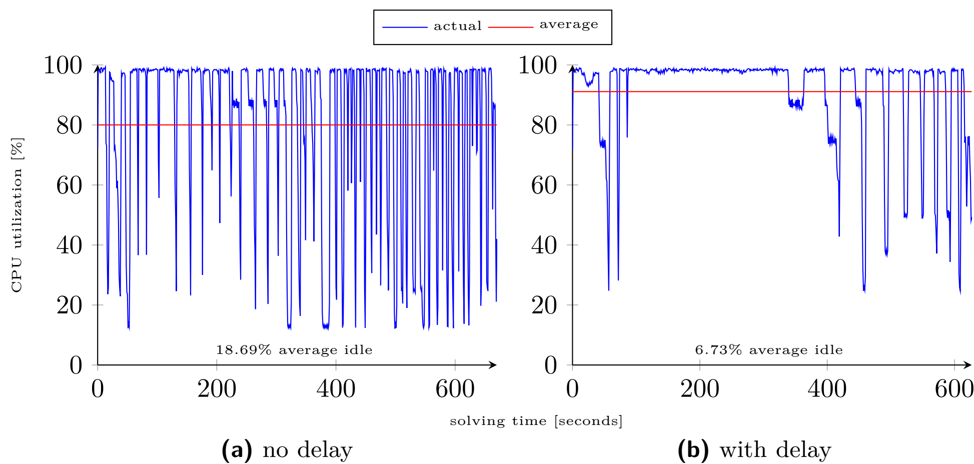
Figure 1 Deterministic clock with different settings.

An ideal deterministic clock closely resembles the CPU time of each thread. The deterministic clock in SCIP was chosen to be a linear combination of solving statistics that are already available in SCIP, such as the number of LP iterations. For the purpose of choosing good coefficients for this linear combination, data was collected from runs on several instances of MIPLIB 2010 [15] with different predefined emphasis settings, e.g. for finding feasible solutions or proving optimality. The data consists of observations of the solving time and the values of the statistics at that time. All the observations collected from the same run are dependent, because the statistics and the solving time are counted from the beginning of the solving process. Hence, the data was transformed by subtracting from each observation the values of the previous observation for each run. The resulting observations pertain to a roughly equal time interval in the solving process, and depend only on solving statistics counted within that interval. Still, predicting the solving time from the statistics across different instances requires to also take the size of a problem into account. Therefore, the observations were scaled by the number of non-zeros in the presolved problem.

To compute the coefficients for the scaled statistics we used a linear regression method called *Lasso* [26], which applies a parametrized ℓ_1-regularization. Penalizing the ℓ_1-norm of the solution vector causes the regression algorithm to bias towards a sparse solution with small coefficients, which aids the prediction accuracy. Other linear regression methods that use a different regularization where also considered, e.g. ℓ_2-regularization or a combination of those. However, the solutions obtained using *Lasso* gave the best predictions.

For the regression an implementation of scikit-learn [20] was used and the amount of regularization was chosen by cross-validation. To avoid overfitting to instance specific traits, the data of a single instance at a time was left out during cross-validation. The resulting linear combination has non-zero coefficients for the number of primal and dual LP iterations with warm-start information, the number of bound changes in probing mode, and the number of calls to an internal function that checks whether the solving has been stopped.

In Figure 1 the deterministic clock is visualized for different instances. The dashed lines show the progression of the deterministic clock with different parameter settings, the solid line shows the wall clock. For the deterministic clock, the emphasis settings for optimality, feasibility and easy instances, that are provided by SCIP, where used in addition to the default settings. The emphasis setting for optimality separates more aggressively, the setting for feasibility applies heuristics more aggressively, and the setting for easy instances avoids expensive techniques for presolving, separation and heuristics to focus on the tree search. The deterministic clock is usually close to the wall clock (Figure 1a and 1b), but on some instances it does not run at the same speed for different settings (Figure 1c). Such a behavior cannot be avoided in general as the deterministic clock is tuned for good results only on average



(a) no delay **(b)** with delay

Figure 2 The CPU utilization of concurrent SCIP using 8 threads on the instance biella1, once without a delay and once using a delay.

and the same setting can produce very different results on different instances. Therefore, a setting on which the deterministic clock runs slower can be detrimental to the performance of concurrent SCIP, even though this setting would be beneficial if the wall clock was used instead.

If solvers are able to access shared information immediately, a *barrier* is required at each communication point, i.e. a point in the program at which a thread can only continue if all other threads have reached the barrier. Because such a barrier-based synchronization scheme would cause a large amount of idle time—especially when the deterministic clock deviates between threads—we introduced a delay before the solvers read data from a communication point. With a delay d, the solvers only read information from communication points that occurred at time $t - d$ or earlier, if their own deterministic clock is at time t. The solvers thereby receive information that is slightly outdated, still, the performance is better because they are waiting to a lesser extent. Even though the solvers still have to wait if the drift of the deterministic clock becomes too large, the CPU utilization improved significantly, as can be seen in Figure 2. Therein the CPU utilization was measured for concurrent SCIP running with different emphasis settings on eight threads. In Figure 2a no delay was used and in Figure 2b the delay was chosen as the deterministic equivalent of one second. Choosing a shorter delay did not make much of a difference here, because the delay effectively amounts at least to the time since the last communication point. With this delay the idle time reduced from 18.69% to 6.73%; put in other words, by using a delay concurrent SCIP was able to utilize roughly one CPU core more.

2.2 FiberSCIP

The Ubiquity Generator (UG) Framework[2] is a framework for the parallelization of branch-and-bound based solvers on distributed or shared memory computing environments. The aim of the UG framework is to parallelize branch-and-bound based solvers from the "outside". In this regard, the UG framework has been used to provide external parallelization for the base solvers SCIP, Xpress [9] and PIPS-SBB [18]. To provide the capability to employ the UG

[2] http://ug.zib.de/

framework on shared and distributed memory environments, two different parallelization implementations are available. The distributed memory implementation of the UG framework uses the standardized Message Passing Interface (MPI). Alternatively, the shared memory implementation makes use of the Pthreads library.

The application of UG to parallelize SCIP has resulted in the solvers FiberSCIP (ug[SCIP, Pthreads]) [25] and ParaSCIP (ug[SCIP, MPI]) [23], for shared and distributed memory respectively. Since FiberSCIP was designed as a development environment for ParaSCIP, it serves as an ideal platform to evaluate the performance of distributed domain propagation and potentially leading to the adoption of this algorithmic feature into a large-scale parallel branch-and-bound solver.

There are three main phases of parallel branch-and-bound based solvers: ramp-up, primary, and ramp-down phases. For details regarding each of these phases, the reader is referred to Ralphs [22], Xu et al. [27], and Shinano et al. [25]. In the current work, the focus will be on the ramp-up phase, which is defined as the time period at the beginning of the solving process until sufficiently many branch-and-bound nodes are available to keep all processing units busy the first time. In the ramp-up phase FiberSCIP provides an implementation of racing ramp-up [25]. At the start of computation this form of ramp-up immediately sends a copy of the root branch-and-bound node to all available threads via the LoadCoordinator and commences parallel solving. To diversify the resulting branch-and-bound trees that are found across the set of all threads, different parameter settings are provided. This form of ramp-up is similar to a portfolio solving approach for MIP.

The different SCIP parameter settings used during racing ramp-up are compiled into FiberSCIP. They are a combination of the emphasis settings provided by SCIP labeled as *off*, *fast*, *default*, and *aggressive* for the different components in SCIP such as primal heuristics, presolvers, and separators. Exactly one solver uses the default settings of SCIP.

3 Distributed domain propagation

Our goal is to exploit variable bound information in a parallel portfolio solver to identify additional domain propagations. We let each solver in a parallel portfolio share new global variable bounds with the other solvers. A solver receiving these bounds propagates them against its local information and again shares the resulting domain reductions with the other solvers. We call this technique *distributed domain propagation* (DDP) and expect it to help solving problems within fewer branch-and-bound nodes, as a result of tighter variable bounds reducing the search space.

Portfolio parallelization involves having different settings in each solver, which results in different solution processes. Notably, each solver may generate conflicts [1] and cuts not generated in any other solver. Also the reduced costs in the LP relaxation of the root node may not be the same due to degeneracy. Since all of this information is used for domain propagation, a bound reduction that can be found in one solver may not be found in the other solvers. As such, DDP is able to perform additional domain reductions in each individual solver by sharing global variable information.

The DDP is implemented on top of the plugin structure of SCIP. It uses an event handler that reacts on global domain reductions for each variable and a propagator that applies the domain reductions received from other solvers. The implementations for concurrent SCIP and FiberSCIP differ in how they transfer the bound from the event handler in one solver to the propagator in another solver.

In concurrent SCIP the event handler stores the best bound for a variable whenever it reacts on a global bound change event. Once a communication point is reached, the bounds

stored in the event handler are passed to a shared data structure where they are merged with bound changes from other solvers. If a solver reads this data structure, all bounds that are tighter than the current ones are passed to the propagator. The next time SCIP does domain propagation it will call the propagator, which will then apply the domain reductions.

In FiberSCIP a different implementation of DDP is provided. The major difference lies in the method of communication. When either a new incumbent solution is found or the domain of a variable has been reduced in a solver, this information is sent to the LoadCoordinator immediately. The LoadCoordinator stores the best incumbent solution and the tightest lower and upper bounds for each variable. When the LoadCoordinator receives an updated solution or bound, it is broadcasted to all solvers immediately. This results in asynchronous communication between all solvers. After receiving the bound, the procedure is the same as that of concurrent SCIP.

SCIP applies so-called dual reductions, which may cut off feasible solutions. Some of these reductions can cut off optimal solutions, but guarantee to keep at least one optimal solution. However, if such reductions from different solvers are applied together, it may happen that all optimal solutions are cut off. Accordingly, these dual reductions are disabled in all but one of the solvers. Thus it is ensured that the variable domains remain valid for all solvers. Note that reduced cost strengthening can be applied in all solvers, since it does not cut off optimal solutions.

Another difficulty for sharing variable bounds is the different formulations that arise from using different presolving techniques in each solver. When transferring a variable bound from one solver to another, it must first be transformed back into the original problem formulation and then re-transformed into the formulation of the solver that receives the bound.

4 Computational results

Computational experiments have been performed with SCIP 4.0.0 using SoPlex 3.0.0 [16] as an LP solver. The time limit was set to one hour on a cluster with 128GB memory and two Intel Xeon E5-2690 v4 2.60GHz processors per node. A subset of instances collected from the test sets of MIPLIB 3.0 [6], MIPLIB 2003 [3], and the benchmark set of MIPLIB 2010 [15] were used for the experiments. The subset was selected by excluding instances that default SCIP solved in less than a second or within the root node. Furthermore, the instances mspp16 and bley_xl1 were excluded due to memory issues and errors in one of the solvers, respectively. The resulting test set contains 125 instances.

The settings used for the different SCIP solvers in concurrent SCIP and in FiberSCIP were the same settings that FiberSCIP uses for racing ramp-up. The default behavior of presolving a problem instance before distributing it to the solvers was disabled. This makes the solving behavior of concurrent SCIP and FiberSCIP closer to that of default SCIP—aiding the comparability of their results.

Table 1 shows a comparison of the number of bounds that where tightened via DDP. For both implementations the number of such domain reductions were counted on all variables and also the subset applied to integer variables. The results are given for the winning solver and were aggregated by using a shifted geometric mean with a shift of 10. An interesting observation is that a larger number of threads leads to more domain reductions being found by DDP. This stems from the effect explained in the previous section, since more solvers with different configurations result in more diverse information being used for domain propagation.

[3] http://miplib.zib.de/

Table 1 Comparison of the number of domain reductions that were found via DDP in concurrent SCIP and FiberSCIP. The domain reductions on the subset of integer variables are given additionally in the second column.

Solver	Settings	#Dom. red.	#Int. dom. red.
Concurrent SCIP	4 threads	15.3	7.6
	8 threads	17.6	7.9
	12 threads	21.9	8.8
Concurrent SCIP (wall clock)	4 threads	16.0	8.7
	8 threads	18.8	9.8
	12 threads	27.9	14.3
FiberSCIP	4 threads	89.9	42.8
	8 threads	130.7	55.9
	12 threads	147.9	60.5

Additionally, the results show a huge difference in the number of domain reductions found by DDP between concurrent SCIP and FiberSCIP which is caused by the different communication schemes; in FiberSCIP a new bound reduction is communicated immediately and will therefore be received with a much smaller delay than in concurrent SCIP. Thus DDP could find a domain reduction that the solvers may have found by themselves shortly after. In concurrent SCIP that is more unlikely since it will communicate less frequently and the other solvers will read the shared domain reductions later, due to the delay used in this implementation. Also, concurrent SCIP will only communicate the best bound of a variable for which SCIP finds subsequent domain reductions between two communication points.

The performance of each portfolio solver with and without DDP is presented in Table 2. In preparing these results, only a subset of the original test set was used that contained 75 instances where at least one bound was tightened by DDP. The reduction of the test set is justified, because DDP does no additional computations and only communicates if domain reductions are found. Hence, it has no measurable impact if there are no domain reductions and including all the instances would merely introduce random noise to the comparison due to the non-deterministic behavior of FiberSCIP. In Table 2 the number of nodes were aggregated with a geometric mean shifted by 100 and the time was aggregated with a geometric mean shifted by 10.

In most settings a positive impact of DDP on the running time and the number of nodes is visible. However, on the 4 thread setting for FiberSCIP and concurrent SCIP using the deterministic clock DDP did not seem to help. Because DDP performed very well with the same settings in concurrent SCIP using the wall clock, we attribute the outlier to performance variability. Also it is expected that the performance variability is larger with a smaller number of threads.

Due to the overhead introduced by the deterministic synchronization, FiberSCIP is expected to outperform concurrent SCIP. Nevertheless, the large difference that also occurs when using the wall clock in concurrent SCIP indicates that the parameters which control the communication need to be adjusted. Notably, the delay and the synchronization frequency seem to be suboptimal. Also it can be observed on both implementations that DDP performs better with fewer than 12 threads. This is only partially caused by an increased communication effort when more solvers are used. More importantly, the architecture of SCIP is not yet exploiting shared memory parallelism and requires to duplicate an unnecessary large amount

Table 2 Comparison of default SCIP, concurrent SCIP using the deterministic clock or the wall clock, and FiberSCIP on the 75 instances that were solved with all settings and where DDP was able to find at least one domain reduction in any setting.

Solver	Settings	with DDP Time	with DDP Nodes	without DDP Time	without DDP Nodes
FiberSCIP	4 threads	119.9	5129.5	118.8	5217.5
	8 threads	112.9	4087.6	120.2	4406.2
	12 threads	121.5	4294.3	123.7	4286.3
Concurrent SCIP	4 threads	172.2	5354.9	172.9	5512.8
	8 threads	179.4	4971.4	182.1	4821.3
	12 threads	202.8	4976.6	205.8	4543.8
Concurrent SCIP (wall clock)	4 threads	136.2	5243.8	143.9	5631.0
	8 threads	140.7	4527.8	145.0	4660.2
	12 threads	152.6	4557.7	155.8	4799.4
SCIP	default			148.2	8556.1

of data and SCIP's performance is already memory bandwidth bound in many parts of the code. An increased number of threads slows down the computations in each thread even without any communication. The reason for this slow down are various effects caused by the increased load on the systems resources, e.g. more context switches, page faults and cache misses. For a detailed discussion we refer to Shinano et al. [25]. In due consideration of these effects we conclude that an increased performance for a larger number of threads is not an issue of the algorithmic approach of DDP, but of an efficient implementation that better exploits a shared memory architecture.

5 Concluding Remarks

This paper has introduced distributed domain propagation (DDP), a technique for finding global variable domain reductions in a parallel portfolio solver. Computational experiments were conducted to compare a deterministic synchronized implementation in concurrent SCIP and an asynchronous implementation in FiberSCIP on standard MIP instances.

In order to reproduce the results presented in this paper the readers can find the source code of SCIP and FiberSCIP in the release version 4.0.0 of the SCIP Optimization Suite (`http://scip.zib.de/#scipoptsuite`) and all instance data that was used can be retrieved from the MIPLIB homepage (`http://miplib.zib.de`).

The computational experiments show that DDP improves the overall performance of a portfolio solver significantly. The communication strategy of FiberSCIP gives better results in the experiments presented here. However, the communication settings and the parameter settings used in the individual solvers are not optimal for concurrent SCIP. Additionally, only concurrent SCIP provides deterministic communication. Both implementations suffer from a general slowdown due to a limited memory bandwidth if more than eight threads are used. For optimal performance one has to strike a balance between the two communication strategies to minimize the communication overhead while domain reductions are still applied within a short time frame. The algorithmic approach of DDP, however, has been shown to significantly improve the performance in a parallel portfolio of MIP solvers.

References

1 Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007. Mixed Integer ProgrammingIMA Special Workshop on Mixed-Integer Programming. doi:10.1016/j.disopt.2006.10.006.

2 Tobias Achterberg. Scip: Solving constraint integer programs. *Math. Prog. Comp.*, 1(1):1–41, 2009.

3 Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. Technical Report 16-44, ZIB, Takustr.7, 14195 Berlin, 2016.

4 Tobias Achterberg and Roland Wunderling. *Mixed Integer Programming: Analyzing 12 Years of Progress*, pages 449–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-38189-8_18.

5 Tomáš Balyo, Peter Sanders, and Carsten Sinz. *HordeSat: A Massively Parallel Portfolio SAT Solver*, pages 156–172. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-24318-4_12.

6 R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, 1998.

7 R. Carvajal, S. Ahmed, G. Nemhauser, K. Furman, V. Goel, and Y. Shao. Using diversification, communication and parallelism to solve mixed-integer linear programs. *Oper. Res. Lett.*, 42(2):186–189, March 2014. doi:10.1016/j.orl.2013.12.012.

8 Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

9 FICO Xpress-Optimizer. http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Optimizer.aspx.

10 Matteo Fischetti, Andrea Lodi, Michele Monaci, Domenico Salvagnin, and Andrea Tramontani. Improving branch-and-cut performance by random sampling. *Mathematical Programming Computation*, 8(1):113–132, 2016. doi:10.1007/s12532-015-0096-0.

11 Gerald Gamrath. Improving strong branching by domain propagation. *EURO Journal on Computational Optimization*, 2(3):99–122, 2014. doi:10.1007/s13675-014-0021-8.

12 Gerald Gamrath, Tobias Fischer, Tristan Gally, Ambros M. Gleixner, Gregor Hendel, Thorsten Koch, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Sebastian Schenker, Robert Schwarz, Felipe Serrano, Yuji Shinano, Stefan Vigerske, Dieter Weninger, Michael Winkler, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 3.2. Technical Report 15-60, ZIB, Takustr.7, 14195 Berlin, 2016.

13 Robert Lion Gottwald. Experiments with a Parallel Portfolio of SCIP Solvers. Master's thesis, Freie Universität Berlin, 2016.

14 Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009. URL: http://jsat.ewi.tudelft.nl/content/volume6/JSAT6_12_Hamadi.pdf.

15 Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010. *Math. Prog. Comp.*, 3:103–163, 2011.

16 Stephen J. Maher, Tobias Fischer, Tristan Gally, Gerald Gamrath, Ambros Gleixner, Robert Lion Gottwald, Gregor Hendel, Thorsten Koch, Marco E. Lübbecke, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Sebastian Schenker, Robert Schwarz, Felipe Serrano, Yuji Shinano, Dieter Weninger, Jonas T. Witt, and Jakob Witzig. The scip optimization suite 4.0. Technical Report 17-12, ZIB, Takustr.7, 14195 Berlin, 2017.

17 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC'01, pages 530–535, New York, NY, USA, 2001. ACM. `doi:10.1145/378239.379017`.

18 Lluis-Miquel Munguia, Geoffrey Oxberry, and Deepak Rajan. Pips-sbb: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs. Technical report, Optimization Online, 2015.

19 George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, NY, USA, 1988.

20 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

21 Ted Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear programming. Technical Report 16-74, ZIB, Takustr.7, 14195 Berlin, 2016.

22 T. K. Ralphs. Parallel branch and cut. In *Parallel Combinatorial Optimization*, pages 53–101. Wiley, 2006.

23 Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. ParaSCIP – a parallel extension of SCIP. In Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010*, pages 135–148. Springer, 2012. `doi:10.1007/978-3-642-24025-6_12`.

24 Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch, and Michael Winkler. Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In *Proc. of 30th IEEE International Parallel & Distributed Processing Symposium*, 2016. to appear.

25 Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. Fiberscip – a shared memory parallelization of scip. Technical Report 13-55, ZIB, Takustr.7, 14195 Berlin, 2013.

26 Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

27 Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Computational experience with a software framework for parallel integer programming. *The INFORMS Journal on Computing*, 21:383–397, 2009.

Efficient Algorithms for k-Regret Minimizing Sets*

Pankaj K. Agarwal[1], Nirman Kumar[2], Stavros Sintos[3], and Subhash Suri[4]

1 **Department of Computer Science, Duke University, Durham, NC, USA**
pankaj@cs.duke.edu
2 **Department of Computer Science, University of Memphis, Memphis, TN, USA**
nkumar8@memphis.edu
3 **Department of Computer Science, Duke University, Durham, NC, USA**
ssintos@cs.duke.edu
4 **Department of Computer Science, UC Santa Barbara, Santa Barbara, CA, USA**
suri@cs.ucsb.edu

Abstract

A regret minimizing set Q is a small size representation of a much larger database P so that user queries executed on Q return answers whose scores are not much worse than those on the full dataset. In particular, a *k-regret minimizing set* has the property that the regret ratio between the score of the top-1 item in Q and the score of the top-k item in P is minimized, where the score of an item is the inner product of the item's attributes with a user's weight (preference) vector. The problem is challenging because we want to find a *single* representative set Q whose regret ratio is small with respect to *all possible* user weight vectors.

We show that k-regret minimization is NP-Complete for all dimensions $d \geq 3$, settling an open problem from Chester et al. [VLDB 2014]. Our main algorithmic contributions are two approximation algorithms, both with provable guarantees, one based on coresets and another based on hitting sets. We perform extensive experimental evaluation of our algorithms, using both real-world and synthetic data, and compare their performance against the solution proposed in [VLDB 14]. The results show that our algorithms are significantly faster and scalable to much larger sets than the greedy algorithm of Chester et al. for comparable quality answers.

1998 ACM Subject Classification H.2.8 Database Applications

Keywords and phrases regret minimizing sets, skyline, top-k query, coreset, hitting set

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.7

1 Introduction

Multi-criteria decision problems pose a unique challenge for databases systems: how to present the space of possible answers to a user. In many instances, there is no single best answer, and often a very large number of incomparable objects satisfy the user's query. For instance, a database query for a car or a smart phone can easily produce an overwhelming number of potential choices to present to the user, with no obvious way to rank them. Top-k and the skyline operators are among the two main techniques used in databases to manage this kind of complexity, but each has its own shortcoming.

* Work by Agarwal and Sintos is supported by NSF under grants CCF-15-13816, CCF-15-46392, and IIS-14-08846, by ARO grant W911NF-15-1-0408, and by Grant 2012/229 from the U.S.-Israel Binational Science Foundation. Work by Suri and Kumar is supported by NSF under grant CCF-15-25817.

© Pankaj K. Agarwal, Nirman Kumar, Stavros Sintos, and Subhash Suri;
licensed under Creative Commons License CC-BY

The top-k operator relies on the existence of a *utility function* that is used to rank the objects satisfying the user's query, and then selecting the top k by score according to this function. A commonly used utility function takes the inner product of the object attributes with a *weight vector*, also called the *user's preference*, thus forming a weighted linear combination of the different features. However, formulating the utility function is complicated, as users often do not know their preferences precisely, and, in fact, exploring the cost-benefit tradeoffs of different features is often the goal of database search.

The second approach of skylines is based on the principle of *pareto optimality*: if an object p is better than another object q on all features, then p is always preferable to q by any rational decision maker. This coordinate-wise dominance is used to eliminate all objects that are dominated by some other object. The *skyline* is the set of objects not dominated by any other object, and has proved to be a powerful tool in multi-criteria optimization. Unfortunately, while skylines are extremely effective in reducing the number of objects in low dimensions, their utility drops off quickly as the dimension (number of features) grows, especially when objects in the database have anti-correlated features (attributes). A related construct called k-skybands [15, 27] grows even more rapidly.

Regret minimization is a recent approach, proposed initially by Nanongkai et al. [31], to address the shortcomings of both the top k and skylines. It hybridizes top k and skylines by computing a small representative subset Q of the much larger database P so that *for any preference vector* the top ranked item in Q is a good approximation of the top ranked item in P. The hope is that the size of Q is much smaller than that of the skyline of P. The goal is to find a subset Q of small size whose approximation error is also small: posed in the form of a decision question, is there a subset of r objects so that every user's top-1 query can be answered within error at most $x\%$? In general, this is too stringent a requirement and motivated Chester et al. [12] to propose a more relaxed version of the problem, called the k-regret minimization.[1] In k-regret minimization, the quality of approximation is measured as the gap between the score of the top 1 item in Q and the top k item in P expressed as a ratio, so that the value is always between 0 and 1.

Problem formulation. An object is represented as a point $p = (p_1, \ldots, p_d)$ in $\mathbb{R}^d$ with non-negative attributes, i.e., $p_i \geq 0$ for every $i \leq d$. Let $\mathbb{X} = \{(p_1, \ldots, p_d) \in \mathbb{R}^d \mid p_i \geq 0 \ \forall i\}$ denote the space of all objects, and let $\mathsf{P} \subset \mathbb{X}$ be a set of n objects. A user preference is also represented as a point $u = (u_1, \ldots, u_d) \in \mathbb{X}$, i.e., all $u_i \geq 0$. Given a preference $u \in \mathbb{R}^d$, we define the *score* of an object p to be $\omega(u, p) = \langle u, p \rangle = \sum_{i=1}^{d} u_i p_i$.

For a preference $u \in \mathbb{X}$ and an integer $k \geq 1$, let $\varphi_k(u, \mathsf{P})$ denote the point p in P with the k-th largest score (i.e., there are less than k points of P with larger score than $\omega(u, p)$ and there are at least k points with score at least $\omega(u, p)$), and let $\omega_k(u, \mathsf{P})$ denote its score. Set $\Phi_k(u, \mathsf{P}) = \{\varphi_j(u, \mathsf{P}) \mid 1 \leq j \leq k\}$ to be the set of k top points with respect to preference u.[2] For brevity, we set $\omega(u, \mathsf{P}) = \omega_1(u, \mathsf{P})$. If P is obvious from the context, we drop P from the list of the arguments, i.e., we use $\omega_k(u)$ to denote $\omega_k(u, \mathsf{P})$ and so on.

---

[1] We should point out that the term k-regret is used to denote different things by Nanongkai et al. [31] and Chester et al. [12]. In the former, k-regret is the representative set of k objects, whereas in the latter, k-regret is used to denote the regret ratio between the scores of top 1 and top k. In our paper, we follow the convention of Chester et al. [12].

[2] If there are multiple objects with score $\omega_j(u, \mathsf{P})$, then either we include all such points in $\Phi_k(u, \mathsf{P})$ or break the tie in a consistent manner.

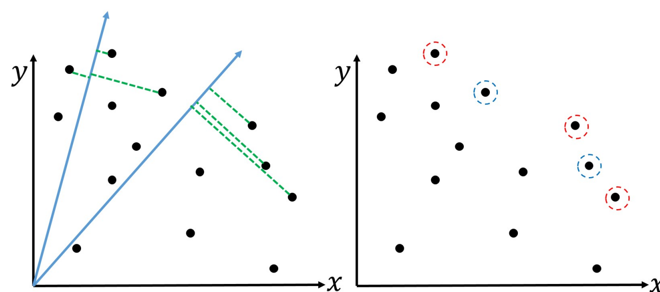


Figure 1 Left: top 3 points in two different preferences. Right: Set of points in the red circles is a $(1,0)$-regret set. Set of points in the blue circles is a $(3,0)$-regret set.

For a subset $\mathsf{Q} \subseteq \mathsf{P}$ and a preference u, define the regret of Q for preference u (w.r.t. P), denoted by $\ell_k(u, \mathsf{Q}, \mathsf{P})$, as

$$\ell_k(u, \mathsf{Q}, \mathsf{P}) = \frac{\max\{0, \omega_k(u, \mathsf{P}) - \omega(u, \mathsf{Q})\}}{\omega_k(u, \mathsf{P})}.$$

That is, $\ell_k(u, \mathsf{Q}, \mathsf{P})$ is the relative loss in the score of the k-th topmost object if we replace P with Q. We refer to the maximum regret of Q, $\ell_k(\mathsf{Q}, \mathsf{P}) = \max_{u \in \mathbb{X}} \ell_k(u, \mathsf{Q}, \mathsf{P})$ as the *regret ratio* of Q (w.r.t. P). If $\ell_k(\mathsf{Q}) \leq \epsilon$, we refer to Q as a (k, ϵ)-regret set (see Figure 1). By definition, a (k, ϵ)-regret set is also a (k', ϵ)-regret set for any $k' \geq k$. In particular, a $(1, \epsilon)$-regret set is a (k, ϵ)-regret set for any $k \geq 1$. However, there may exist a (k, ϵ)-regret set whose size is much smaller than any $(k-1, \epsilon)$-regret set, so the notion of (k, ϵ)-regret set is useful for all k.

Notice that $\ell_k(\mathsf{Q})$ is a monotonic decreasing function of its argument, i.e., if $\mathsf{Q}_1 \subseteq \mathsf{Q}_2$, then $\ell_k(\mathsf{Q}_1) \geq \ell_k(\mathsf{Q}_2)$. Furthermore, for any $t > 0$, $\omega(tu, p) = t\omega(u, p)$ but $\varphi_k(tu, \mathsf{P}) = \varphi_k(u, \mathsf{P})$, $\Phi_k(tu, \mathsf{P}) = \Phi_k(u, \mathsf{P})$, and $\ell_k(tu, \mathsf{Q}, \mathsf{P}) = \ell_k(u, \mathsf{Q}, \mathsf{P})$ (scale invariance).

Our goal is to compute a small subset $\mathsf{Q} \subseteq \mathsf{P}$ with small regret ratio, which we refer to as the *k-regret minimizing set* (k-RMS) problem[3]. Since the regret ratio can be decreased by increasing the size of the subset, there are two natural formulations of the RMS problem.

(i) min-*error*: Given a set P of objects and a positive integer r, compute a subset of P of size r that minimizes the regret ratio, i.e., return a subset $\mathsf{Q}^* = \operatorname{argmin}_{\mathsf{Q} \subseteq \mathsf{P}: |\mathsf{Q}| \leq r} \ell_k(\mathsf{Q})$, where we define $\ell_r = \ell_k(\mathsf{Q}^*)$.

(ii) min-*size*: Given a set P of objects and a parameter $\epsilon > 0$, compute a smallest size subset with regret ratio at most ϵ, i.e., return $\mathsf{Q}^{\#} = \operatorname{argmin}_{\mathsf{Q} \subseteq \mathsf{P}: \ell_k(\mathsf{Q}) \leq \epsilon} |\mathsf{Q}|$, and set $s_\epsilon = |\mathsf{Q}^{\#}|$.

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

(I) In Section 2, we show that the RMS problem is NP-Complete for all dimensions $d \geq 3$ and $k > 1$, answering an open problem from [12]. Previously, the problem was known to be solvable in polynomial time for $d = 2$ and intractable for $d = \Omega(n)$.

(II) In Section 3, we present a coreset-based universal approximation, which shows that every $\mathsf{P} \subset \mathbb{X}$ admits an $O(\frac{1}{\epsilon^{(d-1)/2)}})$ size $(1, \epsilon)$-regret set, and thus also a (k, ϵ)-regret set, for any $k \geq 1$ and $\epsilon > 0$. The size of this regret-set is independent of the size of P, it can be computed in time $O(n + \frac{1}{\epsilon^{d-1}})$, and it can be dynamically updated per point insertion and deletion in time $O(\frac{\text{polylog}(n)}{\epsilon^{d-1}})$.

[3] We will refer to the k-RMS problem simply as RMS problem.

(III) In Section 4, we present an instance-specific approximation scheme, complementing the NP-Completeness of regret-set minimization. This is significant because the size of (k, ϵ)-regret set for a generic P can be much smaller than the coreset-based bound of $1/\epsilon^{\frac{d-1}{2}}$. In particular, our algorithm computes a $(k, 2\epsilon)$-regret set of P[4] whose size is within a log-factor of the optimal (k, ϵ)-regret set. With binary search, we can use this algorithm to also approximate the min-error version of the problem.

(IV) In Section 5, we describe our experimental results and evaluate the efficacy and the efficiency of our algorithms on both synthetic and real data sets. We compare our algorithms with the state of the art greedy algorithm for the k-regret minimization problem presented in [12]. Our hitting-set based algorithm is significantly faster than the previous known algorithms and the maximum regret ratios of the returned sets are very close, if not better, than the maximum regret ratios of the greedy algorithm. The coreset algorithm is significantly faster than hitting set and greedy algorithms. Although the (maximum) regret ratio of the set returned by the core-set based algorithm is worse than those of other algorithms, the regret in 90%–95% directions is roughly the same as that of the other two algorithms.

Remarks. While preparing our submission, we learned of two recent and independent discoveries with partial overlap with our work [8, 4]. In [8] the authors prove that RMS problem is NP-Hard for $k \geq 1$ and $d \geq 3$, and describe a coreset-based approximation algorithm. We present a simpler proof to show that RMS problem is NP-hard for $k \geq 2$ and $d \geq 3$. In addition, we show that RMS problem is NP-Complete which is not straightforward. Our coreset-based algorithm is related to their result, however, we first implemented it and run experiments for the RMS problem comparing the results with other competitive algorithms. In [4] the authors describe an efficient algorithm for 1-RMS problem in 2-d and a near-linear time approximation for the 1-RMS problem in higher dimensions, along with experimental evaluation. We give a more general, randomized approximation algorithm for the k-RMS problem with better approximation factor, and same running time with [4] up to logarithmic factors with high probability.

2 3D RMS is NP-Complete

In this section we show that the k-RMS problem is NP-Complete for $d = 3$ and $k \geq 2$.

Membership in NP. It turns out that due to bit-complexity issues, even establishing membership in NP is not straightforward in $d = 3$, and requires some non-trivial ideas. The starting point of our proof is a polynomial-time algorithm for computing the regret ratio of a subset $\mathsf{Q} \subseteq \mathsf{P}$.

Let $\Omega = \{p - q \mid p, q \in \mathsf{P}, p \neq q\}$ be the set of vectors in directions passing through a pair of points of P. For a vector $w \in \Omega$, let $\mathsf{h}_w : \langle x, w\rangle = 0$, be the plane normal to w passing through the origin. By construction, for $w = p - q$ the score of p is higher than that of q for all preferences in one of the open halfspaces bounded by h_w (namely, $\langle x, w\rangle > 0$), lower in the other halfspace, and equal for all preferences in h_w. Set $\mathsf{H} = \{\mathsf{h}_w \mid w \in \Omega\} \cup \{x_i = 0 \mid 1 \leq i \leq 3\}$, i.e., H includes all the planes h_w along with the coordinate planes. H induces a decomposition $A(\mathsf{H})$ of $\mathbb{R}^3$ into cells of various dimensions, where each cell is a maximal

[4] The approximation ratio 2 is not important. We can actually compute a $(k, t\epsilon)$-regret set for an arbitrary small constant $t > 1$.

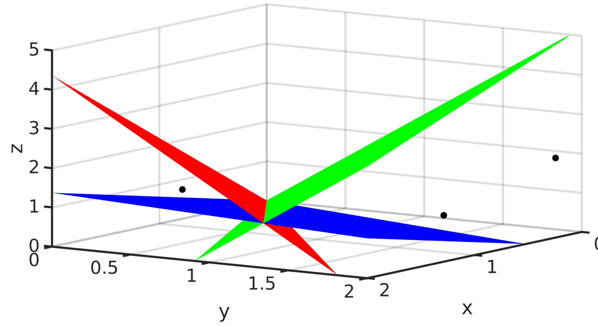


Figure 2 H for a set of 3 points in $\mathbb{R}^3$.

connected region of points lying in the same subset of hyperplanes of H (see Figure 2). It is well known that

(i) each cell of $A(\mathsf{H})$ is a polyhedral cone with the origin as its apex (i.e., each cell is the convex hull of a finite set of rays, each emanating from the origin), and

(ii) the only 0-dimensional cell of $A(\mathsf{H})$ is the origin itself, and the 1-dimensional cells are rays emanating from the origin. Let $\mathcal{C} \subseteq A(\mathsf{H})$ be the set of cells that lie in $\mathbb{X}$, the positive orthant.

For each cell $C \in \mathcal{C}$, let $\ell_k(C, \mathsf{Q}) = \max_{u \in C} \ell_k(u, \mathsf{Q})$ the regret ratio of Q within C. Then $\ell_k(\mathsf{Q}) = \max_{C \in \mathcal{C}} \ell_k(C, \mathsf{Q})$. The following lemma is useful in computing $\ell(C, \mathsf{Q})$.

▶ **Lemma 1.** *For each cell $C \in A(\mathsf{H})$ and for any $i \leq n$, $\varphi_i(u, \mathsf{P})$ (and thus $\varphi_i(u, \mathsf{Q})$) is the same for all $u \in C$.*

Proof. Suppose on the contrary, there are two points $u_1, u_2 \in C$, and $j \geq 0$ such that $\varphi_j(u_1, \mathsf{Q}) \neq \varphi_j(u_2, \mathsf{Q})$. Hence, there are two points p_1, $p_2 \in \mathsf{Q}$ such that $\langle u_1, p_1 \rangle \geq \langle u_1, p_2 \rangle$ and $\langle u_2, p_1 \rangle \leq \langle u_2, p_2 \rangle$, and at least one of the inequalities is strict. Let $\mathsf{h}_w \in \mathsf{H}$ be the plane that is normal to $p_1 - p_2$ and passes through the origin. It divides $\mathbb{R}^3$ into two halfspaces. Reference vectors u_1, u_2 lie in the opposite halfspaces of h_w, and at least one of the u_1, u_2 lies in the open halfspace. However, this is a contradiction because u_1, u_2 lie in the same cell of $A(\mathsf{H})$ and thus lie on the same side of each plane in H. ◀

Fix a cell C. Let $p_i = \varphi_1(u, \mathsf{Q})$ and $p_j = \varphi_k(u, \mathsf{P})$ for any $u \in C$ (from Lemma 1 we have that the ordering inside a cell is the same). Furthermore, let h_j be the plane $\langle x, p_j \rangle = 1$ and let $C^{\downarrow} = \mathsf{h}_j \cap C$. $C^{\downarrow}$ is a 2D polygon and each ray ρ in C intersects $C^{\downarrow}$ at exactly one point $\rho^{\downarrow}$. Since $\ell(u, \mathsf{Q})$ is the same for all points on ρ, $\ell_k(C, \mathsf{Q}) = \ell_k(C^{\downarrow}, \mathsf{Q})$. Furthermore, by Lemma 1, $\ell_k(C^{\downarrow}, \mathsf{Q})$ is either 0 for all $u \in C^{\downarrow}$ or

$$\ell_k(C^{\downarrow}, \mathsf{Q}) = \max_{u \in C^{\downarrow}} \frac{\omega(u, p_j) - \omega(u, p_i)}{\omega(u, p_j)} = \max_{u \in C^{\downarrow}} [1 - \omega(u, p_i)] = 1 - \min_{u \in C^{\downarrow}} \langle u, p_i \rangle.$$

Since $C^{\downarrow}$ is convex and $\langle u, p_i \rangle$ is a linear function, it is a minimum within $C^{\downarrow}$ at a vertex of $C^{\downarrow}$, so we compute $\langle u, p_i \rangle$ for each vertex $u \in C^{\downarrow}$ and choose the one with the minimum value. Repeating this step for all cells of $\mathcal{C}$ we compute $\ell_k(\mathsf{Q})$.

By a well known result in discrete geometry [3], the total number of vertices in $C^{\downarrow}$ over all cells $C \in \mathcal{C}$ is $O(|\mathsf{H}|^2) = O(n^4)$. Furthermore, if b bits are used to represent the coordinates of each point in P, each vertex of $C^{\downarrow}$ requires $O(b)$ bits. Finally, the algorithm extends to higher dimensions in a straightforward manner. The total running time in $\mathbb{R}^d$ is $O(n^{2d-1})$. We thus conclude the following.

▶ **Lemma 2.** *The RMS problem is in* NP.

NP-Hardness Reduction. We first show the hardness for $k = 2$ and $\epsilon = 0$, which is easily extended to other values of k, ϵ. Recall that a preference vector has only non-negative coordinates. For simplicity, however, we first consider all points in $\mathbb{R}^3$ as preference vectors and define $\ell_k(\mathsf{Q}) = \max_{u \in \mathbb{R}^3} \ell_k(u, \mathsf{Q})$, and later we describe how to restrict the preference vectors to $\mathbb{X}$.

Recall that the RMS problem for $\epsilon = 0$ and $k = 2$ asks: Is there a subset $\mathsf{Q} \subseteq \mathsf{P}$ of size r such that in every direction u, the point in Q with the highest score along u, i.e., $\varphi_1(u, \mathsf{Q})$, has score at least as much as that of the second best in P along u, i.e., of the point $\varphi_2(u, \mathsf{P})$?

Let Π be a strictly convex polytope in $\mathbb{R}^3$. The 1-*skeleton* of Π is the graph formed by the vertices and edges of Π. Given Π and an integer $r > 0$, the *convex-polytope vertex-cover* (CPVC) asks whether the 1-skeleton of Π has a vertex cover of size at most r, i.e., whether there is a subset C of vertices of Π of size r such that every edge is incident on at least one vertex of C. The CPVC problem is NP-Complete, as shown by Das and Goodrich [13]. Given Π with V as the set of its vertices, we construct an instance of the RMS problem for $k = 2$, as follows. First we translate Π so that the origin lies inside Π. Next we set $\mathsf{P} = V$. The next lemma proves the NP-hardness of the RMS problem for $k = 2$ and $\epsilon = 0$.

▶ **Lemma 3.** *$\mathsf{Q} \subseteq V$ is a vertex cover of Π if and only if Q is a $(2, 0)$-regret set for P.*

Proof. If Q is a vertex cover of Π, we show that Q is also a $(2, 0)$-regret set. Take a vector $u \in \mathbb{R}^3$ and assume that $q = \varphi_1(u, \mathsf{P})$ (if there is more than one point with rank one, we can let q be any one of them). If $q \in \mathsf{Q}$ then obviously $\omega_1(u, \mathsf{Q}) = \omega_1(u, \mathsf{P}) \geq \omega_2(u, \mathsf{P})$. Now, assume that $q \notin \mathsf{Q}$. Let $(q, q_1), \ldots, (q, q_g)$ be the edges in Π incident on q. Set $N_q = \{q_i \mid 1 \leq i \leq g\}$. Since Q is a vertex cover of Π and $q \notin \mathsf{Q}$, $N_q \subseteq \mathsf{Q}$. We claim that $\varphi_2(u, \mathsf{P}) \in N_q$, which implies that $\omega(u, \mathsf{Q}) \geq \omega_2(u, \mathsf{P})$. Hence, Q is a $(2, 0)$-regret set.

Indeed, since Π is convex, and q is maximal along direction u, the plane h on q vertical to u is a supporting hyperplane for Π. A plane h' parallel to h is translated toward the origin starting with its initial position at h. There are two cases. In the first case, where q and $\varphi_2(u, \mathsf{P})$ have the same score, they belong to the same face of Π that must be contained in h itself – in this case h also contains a point from N_q, since every face containing q and points other than q must contain a 1 dimensional face as well, and therefore a point in N_q. In the second case, as h' is translated, it must first hit one of the neighbors of q, by convexity. As a result, in any case, there will be a point in N_q that gives the rank-two point on u.

Next, if Q is a $(2, 0)$-regret set, we show that Q is a vertex cover of Π. Suppose to the contrary Q is not a vertex cover of Π, i.e., there is an edge (q_1, q_2) in Π but $q_1, q_2 \notin \mathsf{Q}$. Since Π is a strictly convex polytope, there is a plane h tangent to Π at the edge (q_1, q_2) that does not contain any other vertex of Π. If we take the direction u normal to h then $\Phi_2(u, \mathsf{P}) = \{q_1, q_2\}$. If $q_1, q_2 \notin \mathsf{Q}$ then $\omega_1(u, \mathsf{Q}) < \omega_2(u, \mathsf{P})$, which contradicts the assumption that Q is a $(2, 0)$-regret set of P. ◀

By applying an affine transformation to the polytope Π, described in Appendix A, we can show that the RMS problem is NP-hard even when preferences are restricted to $\mathbb{X}$.

Choosing $\epsilon > 0$. While the above suffices to prove the hardness of the RMS problem for $\epsilon = 0$, it is possible that when $\epsilon > 0$ the problem is strictly easier. However, we show the stronger result that the RMS problem is NP-hard even when ϵ is required to be strictly positive. In order to get the NP-hardness of the RMS problem for $\epsilon > 0$ and $k = 2$, we find a small enough strictly positive value of ϵ with bounded bit complexity such that any $(2, \epsilon)$-regret set is also a $(2, 0)$-regret set, and vice versa. For each cell $C \in \mathcal{C}$, we take a

direction $u_C \in C$ and let $\lambda_C = 1 - \omega_3(u_C, \mathsf{P})/\omega_2(u_C, \mathsf{P}) > 0$. By defining $\epsilon = \frac{1}{2}\min_C \lambda_c$ we can conclude the result.

Larger values of k. By making $k-1$ copies of each point in the above construction it is straightforward to show that the RMS problem is NP-complete for any $k \geq 2$ and $d \geq 3$.

▶ **Theorem 4.** *The RMS problem is* NP*-Complete for $d \geq 3$ and for $k \geq 2$.*

3 Coreset-based Approximation

In this section, we present an approximation scheme for the RMS problem using coresets. The general idea of a coreset is to approximately preserve some desired characteristics of the full data set using only a tiny subset [2]. The particular geometric characteristic most relevant to our problem is the *extent* of the input data in any direction, which can be formalized as follows. Given a set of points P and a direction $u \in \mathbb{R}^d$, the *directional width* of P along u, denoted $\mathrm{width}(u, \mathsf{P})$, is the distance between the two supporting hyperplanes of $\mathbb{R}^d$, one in direction u and the other in direction $-u$. The connection between k-regret and the directional width comes from the fact that the supporting hyperplane in a direction u is defined by the extreme point in that direction, and its distance from the origin is simply its score. Therefore, we have the equality: $\mathrm{width}(u, \mathsf{P}) = \omega(u, \mathsf{P}) + \omega(-u, \mathsf{P})$.

We use coresets that approximate directional width to approximate k-regret sets. In particular, a subset $\mathsf{Q} \subseteq \mathsf{P}$ is called an ϵ*-kernel* coreset if $\mathrm{width}(u, \mathsf{Q}) \geq (1-\epsilon)\,\mathrm{width}(u, \mathsf{P})$, for all directions $u \in \mathbb{R}^d$. Showing that an ϵ-kernel coreset of P is also an $(1, \epsilon)$-regret set of P and using the results of [1, 10] to compute small ϵ-kernel coresets efficiently, we prove the following result.

▶ **Theorem 5.** *Given a set P of n points in $\mathbb{R}^d$, $\epsilon > 0$ and an integer $k > 0$, we can compute in time $O(n + \frac{1}{\epsilon^{d-1}})$ a subset $\mathsf{Q} \subseteq \mathsf{P}$ of size $O(\frac{1}{\epsilon^{(d-1)/2}})$ whose k-regret ratio is at most ϵ. The set Q can also be maintained under insertion/deletion of points in P in time $O(\frac{\log^d n}{\epsilon^{d-1}})$ per update.*

Proof. We first show that if $\mathsf{Q} \subseteq \mathsf{P}$ is an ϵ-kernel coreset of P then Q is also $(1, \epsilon)$-regret set of P. If Q is an ϵ-kernel coreset of P then $\mathrm{width}(u, \mathsf{P}) - \mathrm{width}(u, \mathsf{Q}) \leq \epsilon\,\mathrm{width}(u, \mathsf{P}) \leq \epsilon\omega(u, \mathsf{P})$. The last inequality follows because $\omega(-u, \mathsf{P}) \leq 0$. Furthermore $\omega(-u, \mathsf{Q}) \leq \omega(-u, \mathsf{P})$. We thus have

$$\begin{aligned} \omega(u, \mathsf{P}) - \omega(u, \mathsf{Q}) &= \omega(u, \mathsf{P}) + \omega(-u, \mathsf{P}) - \omega(u, \mathsf{Q}) - \omega(-u, \mathsf{P}) \\ &\leq \mathrm{width}(u, \mathsf{P}) - \omega(u, \mathsf{Q}) - \omega(-u, \mathsf{Q}) \\ &= \mathrm{width}(u, \mathsf{P}) - \mathrm{width}(u, \mathsf{Q}) \\ &\leq \epsilon\omega(u, \mathsf{P}). \end{aligned}$$

Hence, $\omega(u, \mathsf{Q}) \geq (1-\epsilon)\omega(u, \mathsf{P})$ and Q is an $(1, \epsilon)$-regret set of P. Chan [10] has described an algorithm for computing ϵ-kernel of size $O(\frac{1}{\epsilon^{(d-1)/2}})$ in time $O(n + \frac{1}{\epsilon^{d-1}})$. The dynamic update performance follows from the construction in [1]. ◀

By comparison, the algorithm in [31] computes a set Q with $\ell_1(Q) \leq \frac{d-1}{(r-d+1)^{\frac{1}{d-1}}+d-1}$, which implies a $(1, \epsilon)$-regret set of size $O(\frac{1}{\epsilon^{d-1}})$. Thus, our result improves the bound of [31] significantly. In addition, if points in P lie uniformly on the unit sphere, then it is known that a valid ϵ-kernel Q can have size $\Omega(\frac{1}{\epsilon^{(d-1)/2}})$, and hence the result in [1] is optimal in the worst case. By a similar construction we can also show that the regret-set bound achieved in Theorem 5 is asymptotically optimal.

4 Regret Approximation using Hitting Sets

While Theorem 5 shows that every point set admits a (k,ϵ)-regret set of size $O(\frac{1}{\epsilon^{(d-1)/2}})$, a specific instance of P may have a much smaller regret set. In this section, we present an efficient scheme to approximate the optimal (k,ϵ)-regret set, by formulating the RMS problem as a hitting-set problem.

A range space (or set system) $\Sigma = (\mathsf{X}, \mathcal{R})$ consists of a set X of objects and a family $\mathcal{R}$ of subsets of X. A subset $H \subseteq \mathsf{X}$ is a hitting set of Σ if $H \cap R \neq \emptyset$ for all $R \in \mathcal{R}$. The hitting set problem asks to compute a hitting set of the minimum size. The hitting set problem is a classical NP-Complete problem, and a well-known greedy $O(\log n)$-approximation algorithm is known.

We construct a set system $\Sigma = (\mathsf{P}, \mathcal{R})$ such that a subset $\mathsf{Q} \subseteq \mathsf{P}$ is a (k,ϵ)-regret set if and only if Q is a hitting set of Σ. We then use the greedy algorithm to compute a small-size hitting set of Σ. A weakness of this approach is that the size of $\mathcal{R}$ could be very large and the greedy algorithm requires $\mathcal{R}$ to be constructed explicitly. Consequently, the approach is expensive even for moderate inputs say $d \sim 5$.

Inspired by the above idea, we propose a bicriteria approximation algorithm: given P and $\epsilon > 0$, we compute a subset $\mathsf{Q} \subseteq \mathsf{P}$ of size $O(s_\epsilon \log s_\epsilon)$ that is a $(k, 2\epsilon)$-regret set of P; the constant 2 is not important, it can be made arbitrarily small at the cost of increasing the running time. By allowing approximations to both the error and size concurrently, we can construct a much smaller range space and compute a hitting set of this range space.

The description of the algorithm is simpler if we assume the input to be well conditioned. We therefore transform the input set, without affecting an RMS, so that the score of the topmost point does not vary too much with the choice of preference vectors, i.e., the ratio $\frac{\max_{u\in\mathbb{X}} \omega(u,\mathsf{P})}{\min_{u\in\mathbb{X}} \omega(u,\mathsf{P})}$ is bounded by a constant that depends on d.

We transform P into another set P', so that (i) for any $u \in \mathbb{X}$, $\varphi_1(u, \mathsf{P}')$ does not lie close to the origin and (ii) for any (k,ϵ)-regret set $\mathsf{Q} \subseteq \mathsf{P}$, the corresponding subset $\mathsf{Q}' \subseteq \mathsf{P}'$ is a (k,ϵ)-regret set in P', and vice versa. The transformation is a non-uniform scaling of P. Nanongkai et al. [31] showed that such a scaling of P satisfies (ii). For each $1 \le j \le d$, let $m_j = \max_{p_i \in P} p_{ij}$ be the maximum value of the jth coordinate among all points. Let $B \subseteq \mathsf{P}$ contains the points corresponding to these m_j values. We refer to B as the *basis* of P. We divide the j-th coordinate of all points by m_j, for all $j = 1, 2, \ldots, d$. Let P' be the resulting set, and let B' be the transformation of B. We note that for each coordinate j there is a point $p'_i \in B'$ with $p'_{ij} = 1$. The different scaling factor in each coordinate can be represented by the diagonal matrix M where $M_{jj} = 1/m_j$, and so $\mathsf{P}' = M\mathsf{P}$. The key property of this affine transformation is the following lemma.

▶ **Lemma 6.** *Let M be the affine transformation described above and let $\mathsf{P}' = M\mathsf{P}$. Then, $\sqrt{d} \cdot \|u\| \ge \omega(u, \mathsf{P}') \ge \frac{1}{\sqrt{d}} \cdot \|u\|$, for all $u \in \mathbb{X}$.*

Proof. Since $\omega(\cdot,\cdot)$ is a linear function, without loss of generality consider a vector $u \in \mathbb{X}$ with $\|u\| = 1$. After the transformation M, for each coordinate j, we have $p'_j \le 1$. Therefore, $\|p'\| \le \sqrt{d}$ and also $\sqrt{d} \ge \omega(u, \mathsf{P}')$ because u is a unit vector. For the second inequality, we note that for any unit norm vector u we must have $u_j \ge \frac{1}{\sqrt{d}}$, for some j. Since our transform ensures the existence of a point $p' \in B'$ with $p'_j = 1$, we must have $\omega(u, \mathsf{P}') \ge \langle u, p' \rangle \ge \frac{1}{\sqrt{d}}$. This completes the proof. ◀

4.1 Approximation Algorithms

We first show how to formulate the min-size version of the RMS problem as a hitting set problem. Let P, k, and ϵ be fixed. For a vector $u \in \mathbb{X}$, let $R_u = \{p \in \mathsf{P} \mid \omega(u,p) \geq (1-\epsilon)\omega_k(u,p)\}$. Note that if $\epsilon = 0$, then $R_u = \Phi_k(u)$, the set of top-k points of P in direction u. Set $\mathcal{R}_u = \{R_u \mid u \in \mathbb{X}\}$. Although there are infinitely many preferences we show below that $|\mathcal{R}_u|$ is polynomial in $|\mathsf{P}|$. We now define the set system $\Sigma = (\mathsf{P}, \mathcal{R}_u)$.

▶ **Lemma 7.**
(i) $|\mathcal{R}_u| = O(n^d)$.
(ii) *A subset* $\mathsf{Q} \subseteq \mathsf{P}$ *is a hitting set of* Σ *if and only if* Q *is a* (k,ϵ)*-regret set of* P.

Proof.
(i) Note that R_u is a subset of P that is separated from $\mathsf{P} \setminus R_u$ by the hyperplane $\mathsf{h}_u : \langle u, x\rangle \geq (1-\epsilon)\omega_k(u,\mathsf{P})$. Such a subset is called linearly separable. A well-known result in discrete geometry [3] shows that a set of n points in $\mathbb{R}^d$ has $O(n^d)$ linearly separable subsets. This completes the proof of (i).
(ii) First, by definition any (k,ϵ)-regret set Q has to contain a point of R_u for all $u \in \mathbb{X}$ because otherwise $\ell_k(u,\mathsf{Q}) > \epsilon$. Hence, Q is a hitting set of Σ. Conversely, if $\mathsf{Q} \cap R_u \neq \emptyset$, then $\ell_k(u,\mathsf{Q}) \leq \epsilon$. If Q is a hitting set of Σ, then $\mathsf{Q} \cap R_u \neq \emptyset$ for all $u \in \mathbb{X}$, so Q is also a (k,ϵ)-regret set. ◀

We can thus compute a small-size (k,ϵ)-regret set of P by running the greedy hitting set algorithm on Σ. In fact, the greedy algorithm in [7] returns a hitting set of size $O(s_\epsilon \log s_\epsilon)$. As mentioned above, the challenge is the size of $\mathcal{R}_u$. Even for small values of k, $|\mathcal{R}_u|$ can be $\Omega(n^{\lfloor d/2 \rfloor})$ [3]. Next, we show how to construct a much smaller set system.

Recall that $\ell_k(u,\mathsf{Q})$ is independent of $\|u\|$ so we focus on unit preference vectors, i.e., we assume $\|u\| = 1$. For the analysis, we also assume that $\mathsf{P} = M\mathsf{P}$ from Lemma 6. Let $\mathbb{U} = \{u \in \mathbb{X} \mid \|u\| = 1\}$ be the space of all unit preference vectors; $\mathbb{U}$ is the portion of the unit sphere restricted to the positive orthant. For a given parameter $\delta > 0$, a set $\mathsf{N} \subset \mathbb{U}$ is called a δ*-net* if the spherical caps of radius δ around the points of N cover $\mathbb{U}$, i.e. for any $u \in \mathbb{U}$, there is a point $v \in \mathsf{N}$ with $\langle u, v\rangle \geq \cos(\delta)$. A δ-net of size $O(\frac{1}{\delta^{d-1}})$ can be computed by drawing a "uniform" grid on $\mathbb{U}$. In practice, it is simpler and more efficient to choose a random set of $O(\frac{1}{\delta^{d-1}} \log \frac{1}{\delta})$ directions – this will be a δ-net with probability at least 1/2. Let N be a $\frac{\delta}{2d}$-net of $\mathbb{U}$, and let $\mathcal{R}_\mathsf{N} = \{R_u \mid u \in \mathsf{N}\}$.

Set $\Sigma_\mathsf{N} = (\mathsf{P}, \mathcal{R}_\mathsf{N})$. Note that $|\mathcal{R}_\mathsf{N}| = O(\frac{1}{\delta^{d-1}})$. Our main observation, stated in the lemma below and proven in Appendix B, is that it suffices to compute a hitting set of Σ_N. Recall that basis B of P is the subset of at most d points, one per coordinate, corresponding to the points with the highest value per coordinate.

▶ **Lemma 8.** *Let* Q' *be a hitting set of* Σ_N*, and let* B *be the basis of* P*. Then* $\mathsf{Q} = \mathsf{Q}' \cup B$ *is a* $(k, \epsilon + \delta - \delta\epsilon)$*-regret set of* P.

Algorithm 1 summarizes the algorithm. Greedy_HS is the greedy algorithm in [7] for computing a hitting set. Scale(P) is the procedure that scales the set P according to the transformation M in Lemma 6. Basis(P) is the method to find the basis B.

Analysis. The correctness of the algorithm follows from Lemma 8. Since a hitting set of Σ is also a hitting set of Σ_N, Σ_N has a hitting set of size at most s_ϵ. The greedy algorithm in [7] returns a hitting set of size $O(s_\epsilon \log s_\epsilon)$ for $d \geq 4$ and of size $O(s_\epsilon)$ for $d \leq 3$. Therefore $|\mathsf{Q}| = O(s_\epsilon \log s_\epsilon)$ for $d \geq 4$ and $O(s_\epsilon)$ for $d = 3$. Computing the set B takes $O(n)$ time. N

Algorithm 1 RMS_HS

Input: P: Input points, $k \geq 1$: rank, $\epsilon, \delta \in [0,1]$: error parameters.
Output: Q a $(k, \epsilon + \delta - \delta\epsilon)$-regret set

1: $B := \textsc{Basis}(\mathsf{P})$
2: $\mathsf{P} := \textsc{Scale}(\mathsf{P})$
3: $\mathsf{N} := \frac{\delta}{2d}$-net of $\mathbb{U}$
4: $R_u := \{p \in \mathsf{P} \mid \omega(u,p) \geq (1-\epsilon)\omega_k(u,\mathsf{P})\}$
5: $\mathcal{R}_\mathsf{N} := \{R_u \mid u \in \mathsf{N}\}$
6: $\mathsf{Q}' := \textsc{Greedy_HS}(\mathsf{P}, \mathcal{R}_\mathsf{N})$
7: Return $\mathsf{Q} := \mathsf{Q}' \cup B$

can be constructed in $O(|\mathsf{N}|)$ time and we can compute R_u for each $u \in \mathsf{N}$ in $O(n)$ time. The greedy algorithm in [7] takes $O(\frac{n}{\delta^{d-1}} \log n \log \frac{1}{\delta})$ expected time (the bound on the running time also holds with high probability).

Putting everything together and setting $\delta = \epsilon$, we obtain the following:

▶ **Theorem 9.** *Let $\mathsf{P} \subset \mathbb{X}$ be a set of n points in $\mathbb{R}^d$, $k \geq 1$ an integer, and $\epsilon > 0$ a parameter. Let s_ϵ be the minimum size of a (k,ϵ)-regret set of P. A subset $\mathsf{Q} \subseteq \mathsf{P}$ can be computed in $O\left(\frac{n}{\epsilon^{d-1}} \log(n) \log\left(\frac{1}{\epsilon}\right)\right)$ expected time such that Q is a $(k, 2\epsilon)$-regret set of P. The size of Q is $O(s_\epsilon \log s_\epsilon)$ for $d \geq 4$ and $O(s_\epsilon)$ for $d \leq 3$.*

Notice that we can improve the running time of Theorem 9 to $O\left(\frac{n}{\epsilon^{d-1}}\right)$ using the simple greedy algorithm for the hitting set problem. In this case, the size of Q is $O(s_\epsilon \log \frac{1}{\epsilon})$.

min-error RMS. Recall that the min-error problem takes as input a parameter r, and returns a subset $\mathsf{Q} \subseteq \mathsf{P}$ of size at most r such that $\ell_k(\mathsf{Q}) \leq \ell_r$, where ℓ_r is the minimum regret ratio of a subset of P of size at most r. We propose a bicriteria approximation algorithm for the min-error problem using Algorithm 1.

Let $E = \{1 - \omega_j(u,\mathsf{P})/\omega_k(u,\mathsf{P}) \mid k < j \leq n, u \in \mathsf{N}\}$, and let $\epsilon_0 \in E$ be the smallest value such that $s_{\epsilon_0} \leq r$. Since N is a $\frac{\delta}{2d}$-net it can be shown (similar to the proof of Lemma 8) that $\varepsilon_0 \leq (1-\delta)\ell_r + \delta$, so we can solve the min-error problem approximately by performing a binary search on the values in E. However, testing whether $s_\epsilon \leq r$ for a given ϵ is hard, so we use an approximate decision procedure as follows: For a value $\epsilon \in E$, we run Algorithm 1. If it returns a subset of size larger than $cr \log r$, where $c > 0$ is an appropriate constant, we search among the values larger than ε, and among the smaller values otherwise. In the end, we return a set Q of size $O(r \log r)$. Notice that if $\epsilon > \ell_r$, then Algorithm 1 always returns a set of size less than $cr \log r$. The following theorem summarizes the results of the min-error version of the problem.

▶ **Theorem 10.** *Let $\mathsf{P} \subset \mathbb{X}$ be a set of n points in $\mathbb{R}^d$, $k \geq 1$ an integer, and $r > 0$, $0 < \delta < 1$ two parameters. A subset $\mathsf{Q} \subseteq \mathsf{P}$ can be computed in $O(\frac{n}{\delta^{d-1}} \log(n) \log\left(\frac{1}{\delta}\right) \log\left(\frac{n}{\delta}\right))$ expected time such that $\ell_k(\mathsf{Q}) \leq (1-\delta)\ell_r + \delta$. The size of Q is $O(r \log r)$ for $d \geq 4$ and $O(r)$ for $d \leq 3$.*

Table 1 Summary of datasets used in experiments.

ID	Description	d	n	Skyline
BB	Basketball	5	21961	200
ElNino	Oceanographic	5	178080	1183
Colors	Colors	9	68040	674
AntiCor	Anti-correlated points	4	10000	657
Sphere	Points on unit sphere	4	15000	15000
SkyPoints	Many points close to skyline	3	500	100

5 Experiments

We have implemented our algorithms as well as the current state of the art, namely, the greedy algorithms described in [31, 12], and experimentally evaluated their relative performance.[5]

Algorithms. In particular, the four algorithms we evaluate are the following:

RRS is the ***R***andomized ***R***egret ***S***et algorithm, based on coresets, described in Section 3. In our implementation, rather than choosing $O(\frac{1}{\epsilon^{(d-1)/2}})$ random preferences all at once, we choose them in stages and maintain a subset Q until $\ell_k(\mathsf{Q}) \leq \epsilon$.

HS is the ***H***itting ***S***et algorithm presented in Section 4. Notice that for $k = 1$, the optimum solution of the RMS problem will always be a subset of the skyline of P. Hence, to reduce the running time we only run the algorithm for $k = 1$ on skyline points. Furthermore, instead of choosing $O(\frac{1}{\epsilon^{d-1}})$ directions in one step and find a hitting set, we can sample in stages and maintain a hitting set until we find a (k, ϵ)-regret set.

NSLLX is the greedy algorithm for 1-RMS problem described in [31], which iteratively finds the preference u with the maximum regret using an LP algorithm and adds $\varphi_1(u, \mathsf{P})$ to the regret set. We use Gurobi software [17] to solve the LP problems efficiently. We remark that this algorithm, as a preprocessing step, removes all data points that are not on the skyline.

CTVW is the extension of the NSLLX algorithm for $k > 1$, proposed by [12], and it is the state of the art for the k-RMS problem. In [12] they discard all the points not on the skyline as preprocessing to run the experiments. The CTVW algorithm solves many (in the worst-case, $\Omega(n)$) instances of large LP programs to add the next point to the regret set. The number of LP programs is controlled by a parameter T – a larger T increases the probability of adding a good point to the regret set, but also leads to a slower algorithm. In the original paper, the authors suggested a value of T that is exponential in k; for instance, $T \geq 2.4 \times 10^7$ for $k = 10$, which is clearly not practical. In practice, Chester et al. [12] used $T = 54$ for $k = 4$, which is also the value we adopted in our experiments for comparison. Indeed, using $T > 54$ increases the running time but does not lead to significantly better regret sets.

The algorithms are implemented in C++ and we run on a 64-bit machine with four 3600 MHz cores and 16GB of RAM with Ubuntu 14.04. In evaluating the quality $\ell_k(\mathsf{Q})$ of a regret set $\mathsf{Q} \subseteq \mathsf{P}$, we compute the regret for a large set of random preferences (for example for $d = 3$ we take 20000 preferences), and use the maximum value found as our estimate. In fact, this approach gives us the distribution of the regret over the entire set of preference vectors.

[5] All data sets that we used and our implementation can be found on `https://users.cs.duke.edu/~ssintos/kRMS_SEA`

Datasets. We use the following datasets in our experiments, which include both synthetic and real-world.

BB (`databasebasketball.com`) is the basketball dataset that has been widely used for testing algorithms for skyline computation, top-k queries, and the k-RMS problem, [12, 21, 24, 25, 39]. Each point in this dataset represents a basketball player and its coordinates contain five statistics (points, rebounds, blocks, assists, fouls) of the player.

ElNino (`archive.ics.uci.edu/ml/datasets/El+Nino`) is the ElNino dataset containing oceanographic data such as wind speed, water temperature, surface temperature etc. measured by buoys stationed in the Pacific ocean, and also used in [12]. This dataset has some missing values, which we have filled in with the minimum value of a coordinate for the point. If some values are negative they are replaced by the absolute value.

Colors (`www.ics.uci.edu/ mlearn/MLRepository.html`) is a data set containing the mean, standard deviation, and skewness of each H, S, and V in the HSV color space of a color image. This set is also a popular one for evaluating skylines and regret sets (see [5, 31]).

AntiCor is a synthetic set of points with *anti-correlated* coordinates. Specifically, let h be the hyperplane with normal $\mathsf{n} = (1, \ldots, 1)$, and at distance 0.5 from the origin. To generate a point p, we choose a random point $\widetilde{p}$ on $\mathsf{h} \cap \mathbb{X}$, a random number $t \backsim \mathcal{N}(0, \sigma^2)$, for a small standard deviation σ, and $p = \widetilde{p} + tn$. If $p \in \mathbb{X}$ we keep it, otherwise discard p. By design, many points lie on the skyline and the top-k elements can differ significantly even for nearby preferences. This data set is also widely used for testing top-k queries or skyline computation (see [6, 31, 39, 29]). For our experiments we set $\sigma = 0.1$ and generate 10000 points.

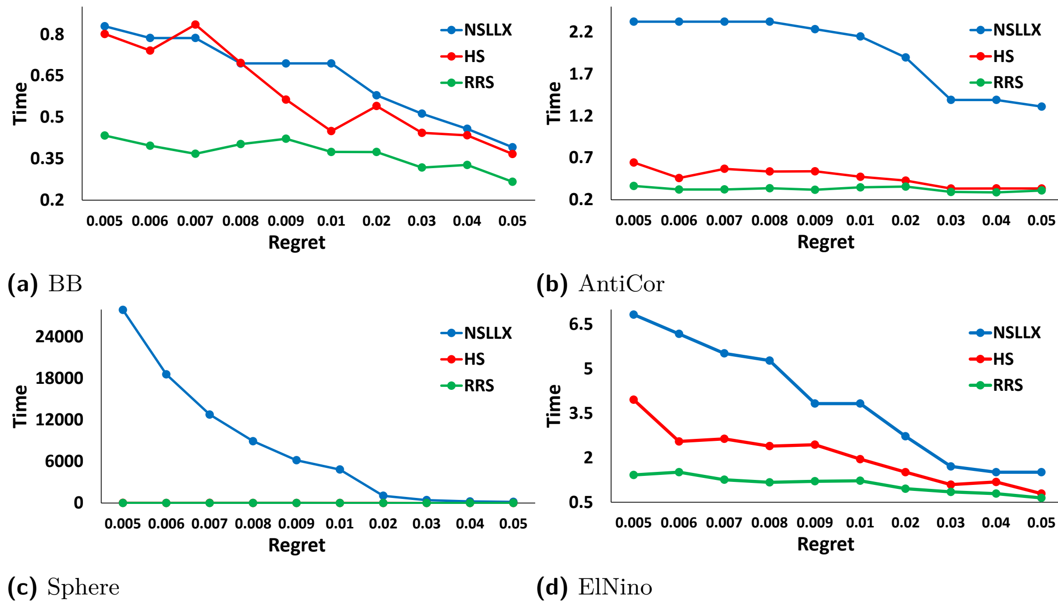
Sphere is a set of points uniformly distributed on the unit sphere inside $\mathbb{X}$, in which clearly all the points lie on the skyline. We generate the Sphere dataset with 15000 points for $d = 4$ (all points lie on the skyline).

SkyPoints is a modification of the Sphere data set. We choose a small fraction of points from the Sphere data set and for each point p add, say, 20 points that lie very close to p but are dominated by p. For larger value of k, say $k > 5$, considering only the skyline points is hard to decide which point is going to decrease the maximum regret ratio in the original set. We generate SkyPoints data set for $d = 3$, 500 points; with 100 points on the skyline.

In evaluating the performance of algorithms, we focus on two main criteria, the runtime and the regret ratio, but also consider a number of other factors that influence their performance such as the value of k, the size of the skyline etc.

RRS and HS are both randomized algorithms so we report the average size of the regret sets and the average running time computed over 5 runs. For $k = 1$, we use the NSLLX algorithm, and for $k = 10$, we use its extension, the CTVW algorithm. In some plots there are missing values for the CTVW algorithm, because we stopped the execution after running it on a data set for 2 days.

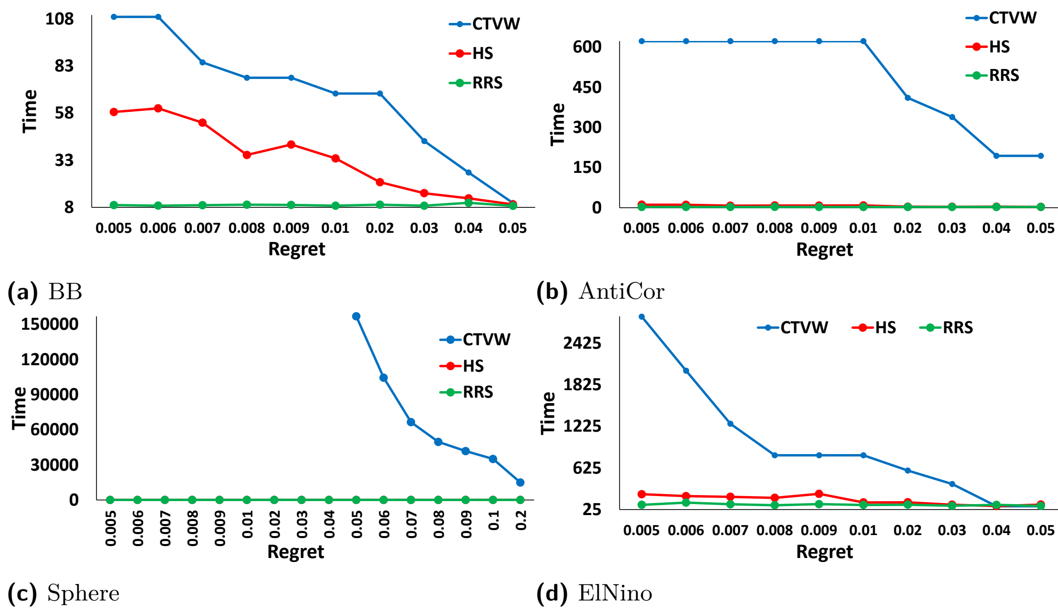
Running time. We begin with the runtime efficiency of the four algorithms, which is measured in the number of seconds taken by each to find a regret set, given a target regret ratio. Figure 3 shows the running times of NSLLX, HS, and RRS for $k = 1$. The algorithm RRS is the fastest. For some instances, the running time of HS and RRS are close but in some other instances HS is up to three times slower. The NSLLX algorithm is the slowest, especially for smaller values of the regret ratio. The relative advantage of our algorithms is quite significant for datasets that have large skylines, such as AntiCor and Sphere. Even for $k = 1$, NSLLX is 7 times slower than HS on AntiCor data set and 480 times slower on Sphere data set, for regret ratio ≤ 0.01.



(a) BB (b) AntiCor

(c) Sphere (d) ElNino

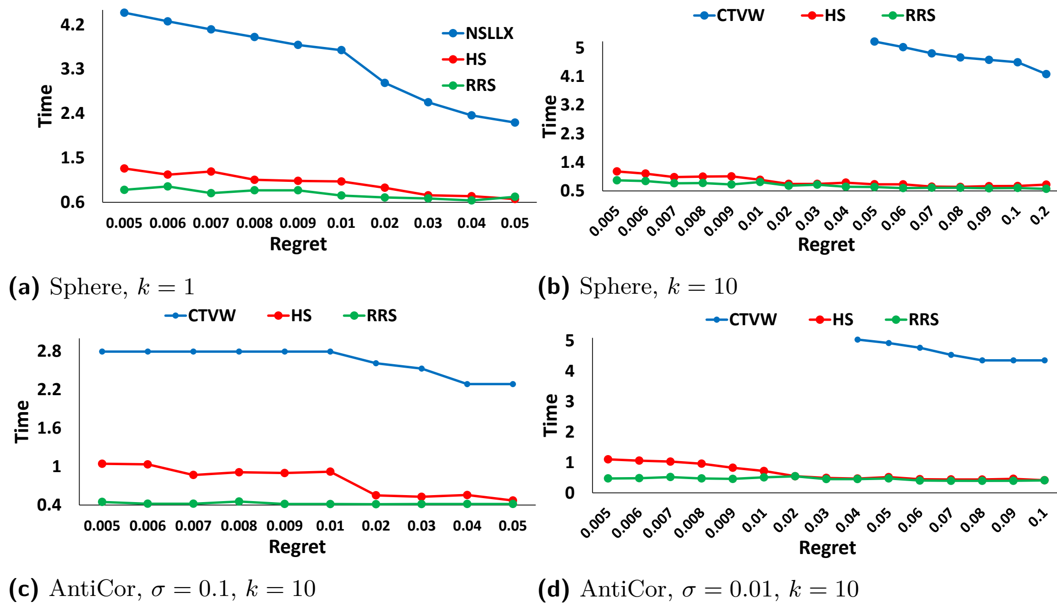
Figure 3 Running time for $k = 1$.



(a) BB (b) AntiCor

(c) Sphere (d) ElNino

Figure 4 Running time for $k = 10$.



(a) Sphere, $k = 1$ **(b)** Sphere, $k = 10$

(c) AntiCor, $\sigma = 0.1$, $k = 10$ **(d)** AntiCor, $\sigma = 0.01$, $k = 10$

Figure 5 $\log_{10}$-scale running time.

The speed advantage of RRS and HS algorithms over CTVW becomes much more pronounced for $k = 10$, as shown in Figure 4. Recall that CTVW discards all points that are not on the skyline. The running time is significantly larger if one runs this algorithm on the entire point set or when the skyline is large. For example, for the AntiCor and Sphere data sets, which have large size skylines, the CTVW algorithm is several orders of magnitude slower than ours. If we set the parameter $\sigma = 0.01$ for AntiCor data set, and generate 10000 points (the skyline has 8070 points in this case) the running time of CTVW is much higher as can be seen in Figure 9b. Because of the high running time of NSLLX and CTVW algorithms, in Figure 5, we show the running time in the log scale with base 10.

Regret ratio. We now compare the quality of the regret sets (size) computed by the four algorithms. Figures 6 and 7 show the results for $k = 1$ and for $k = 10$, respectively.

The experiments show that in general the HS algorithm finds regret sets comparable in size to NSLLX and CTVW. This is also the case for AntiCor data set if we set $\sigma = 0.01$ as can be seen in Figure 9a. The RRS algorithm tends to find the largest regret set among the four algorithms, but it does have the advantage of dynamic udpates: that is, RRS can maintain a regret set under insertion/deletion of points. However, since the other algorithms do not allow efficient updates, we do not include experiments on dynamic updates.

The sphere data set is the worst-case example for regret sets since every point has the highest score for some direction. As such, the size of the regret set is much larger than for the other data sets. HS and RRS algorithm rely on random sampling on preference vectors instead of choosing vectors adaptively to minimize the maximum regret, it is not surprising that for Sphere data sets CTVW does 1.5-3 times better than the HS algorithm. Nevertheless, as we will see below the regret of HS in 95% directions is close to that of CTVW.

Regret distribution. The regret ratio only measures the *largest* relative regret over all preference vectors. A more informative measure could be to look at the entire distribution of the regret over all preference vectors.

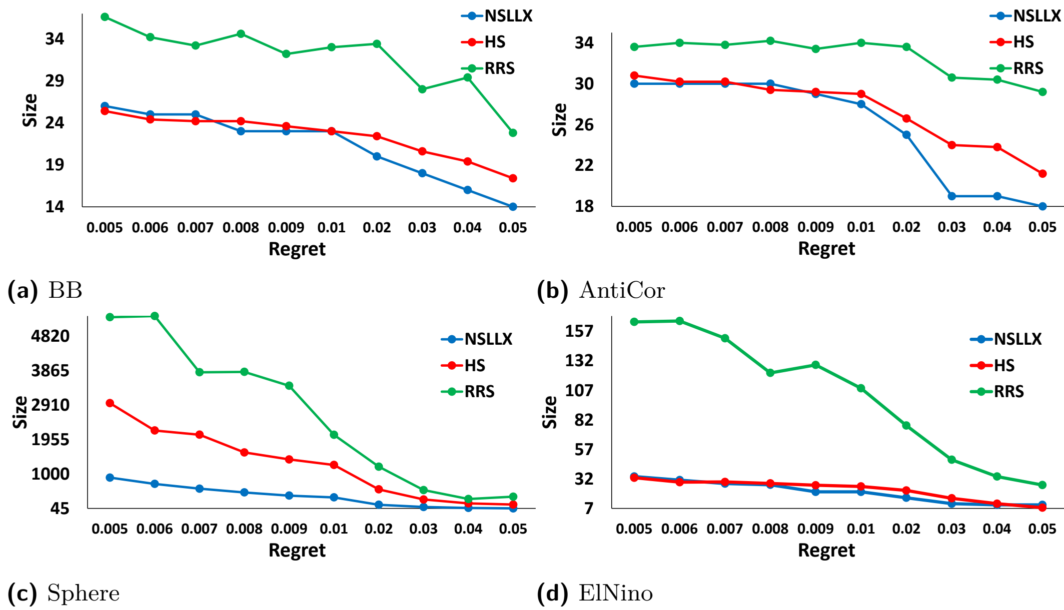


Figure 6 Maximum regret ratio for $k = 1$.

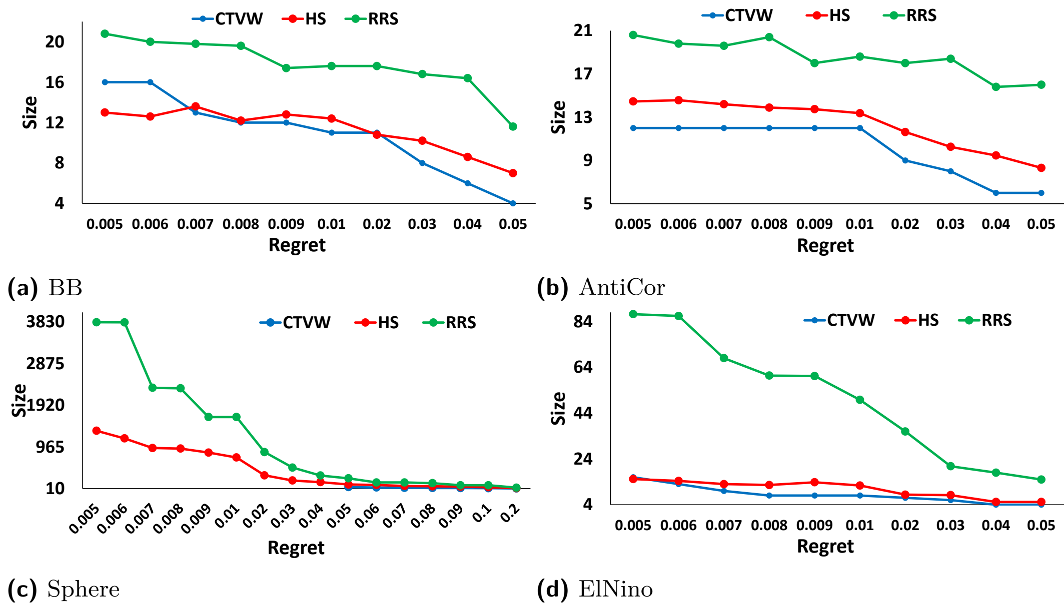


Figure 7 Maximum regret ratio for $k = 10$.

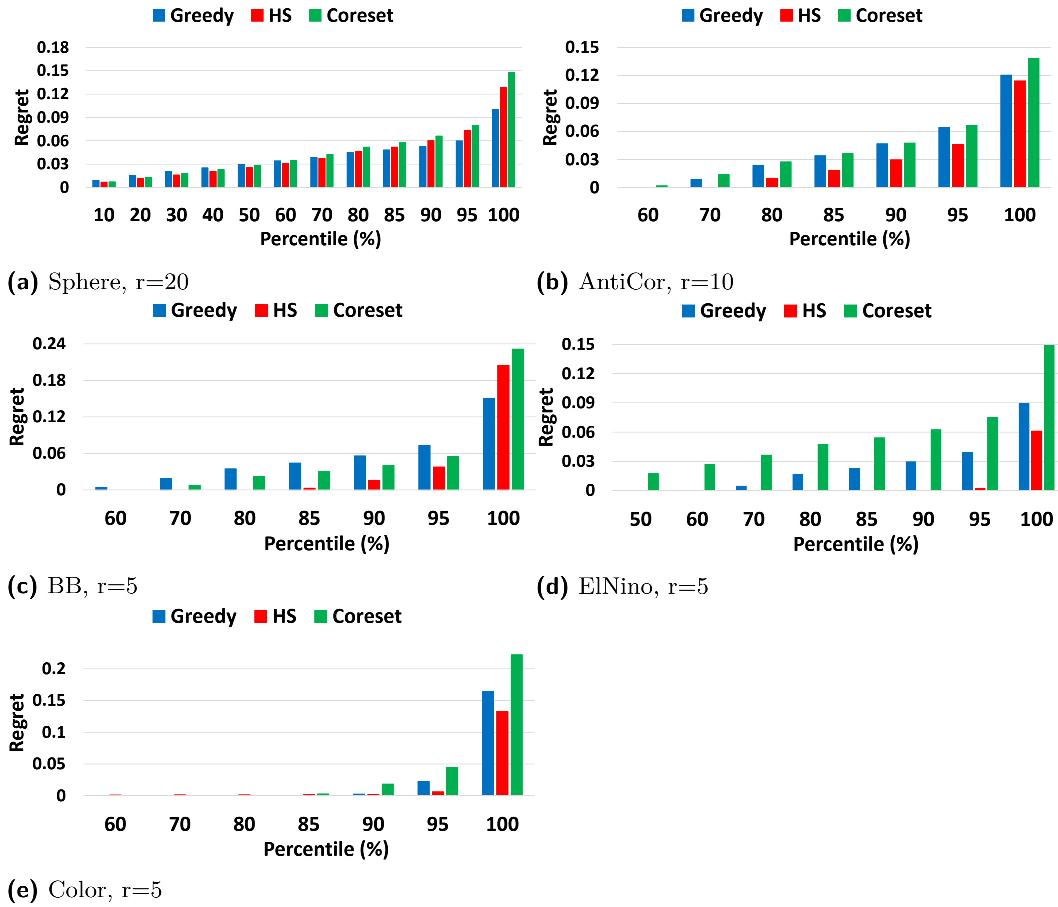
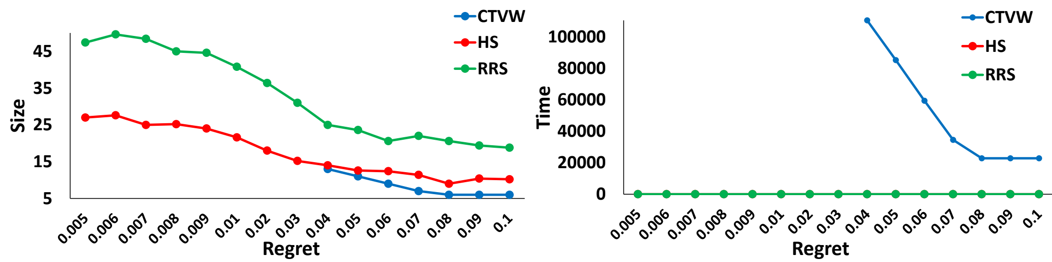


Figure 8 Regret distributions, $k = 1$.

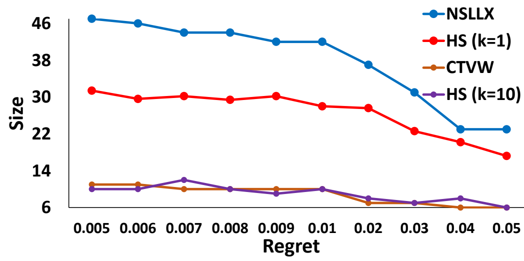
We first show the results for the two synthetic main data sets, namely, Sphere and AntiCor. See Figures 8a, 8b. In this experiment, we fixed the regret set size to 20 for the Sphere dataset and 10 for the AntiCor dataset. We observe that the differences in the regret ratios in 95% of the directions are much smaller than the differences in the maximum regret ratios. For example, the difference of the maximum regret ratio between RRS and NSLLX in Sphere data set is 0.048, while the difference in the 95% (85%) of the directions is 0.019 (0.0096). As we can see in Figures 8c, 8d, 8e we get similar results for the real data sets. For the real datasets we fix the regret size to 5 because for higher values we found that 95% of the directions had 0-regret ratio for all algorithms.

Impact of larger k. We remarked in the introduction that the size of (k, ϵ)-regret set can be smaller for some datasets than their $(1, \epsilon)$-regret set, for $k > 1$. We ran experiments to confirm this phenomenon, and the results are shown in Figures 9c, 9d for Colors data set. As Figure 9c shows, the size of 1-regret set is 3.5 times larger than 10-regret sets for some values of the regret ratio. Figure 9d shows how the size of the regret set computed by the HS algorithm decreases with k, for a fixed value of the regret ratio 0.01.

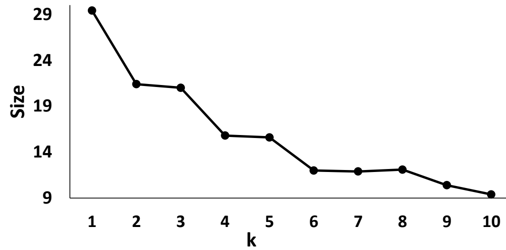
Skyline effect. In order to improve its running time, the algorithm CTVW [12] removes all the non-skyline points, as a preprocessing step, before computing the regret set. While expedient, this strategy also risks finding directions with high k-regret ratio, and as a result



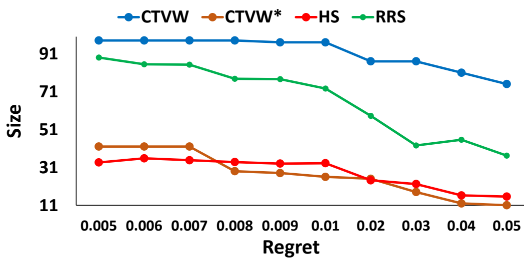
(a) Regret Ratio, $\sigma = 0.01$, $k = 10$ **(b)** Running Time, $\sigma = 0.01$, $k = 10$



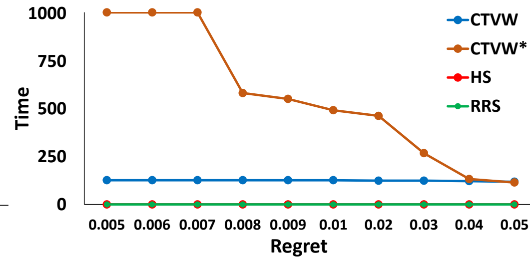
(c) Regret ratio for $k = 1, 10$.



(d) Size of the $(k, 0.01)$-regret set as a function of k.



(e) Regret ratio, SkyPoints



(f) Running time, SkyPoints

Figure 9 Figures 9a, 9b: AntiCor. Figures 9c, 9d: Colors. Figures 9e, 9f: SkyPoints.

may lead to worse regret set. In this experiment, we used the Skypoint dataset to explore this cost/benefit tradeoff. In particular, the modified version of CTVW that does not remove non-skyline points is called CTVW*. The results are shown in Figure 9e, which confirm that removal of non-skyline points can cause significant increase in the size of the regret set, for a given target regret ratio. (In this experiment, the regret size differences are most pronounced for small values of regret ratio. When large values of regret ratio are acceptable, the loss of good candidate points is no longer critical.) Of course, while CTVW* finds nearly as good a regret set as HS, its running time is much worse than that of HS, or CTVW, because of this change, as shown in Figure 9f.

6 Related Work

The work on regret minimization was inspired by preference top-k and skyline queries. Both of these research topics try to help a user find the "best objects" from a database. Top-k queries assign scores to objects by some method, and return the objects with the topmost k scores while the skyline query finds the objects such that no other object can be strictly better. Efficiently answering top-k queries has seen a long line of work, see e.g. [14, 16, 18, 19, 26, 28, 35, 36, 40, 41, 43] and the survey [20]. In earlier work, the ranking

of points was done by weight, i.e., ranking criterion was fixed. Recent work has considered the specification of the ranking as part of the query. Typically, this is specified as a preference vector u and the ranking of the points is by linear projection on u see e.g. [14, 19, 41]. Another ranking criterion is based on the distance from a given query point in a metric space i.e., the top-k query is a k-nearest neighbor query [37].

In general, preference top-k queries are hard, and this has led to approximate query answering [11, 41, 42]. Motivated by the need of answering preference top-k queries, Nanongkai et. al. [31] introduced the notion of a 1-regret minimizing set (RMS) query. Their definition attempted to combine preference top-k queries and the concept of skylines. They gave upper and lower bounds on the regret ratio if the size of the returned set is fixed to r. Moreover, they proposed an algorithm to compute a 1-regret set of size r with regret ratio $O\left(\frac{d-1}{(r-d+1)^{1/(d-1)}+d-1}\right)$, as well as a greedy heuristic that works well in practice.

Chester et. al. [12] generalized the definition of 1-RMS to the k-RMS for any $k \geq 1$. They showed that the k-RMS problemis NP-hard when the dimension d is also an input to the problem, and they provided an exact polynomial algorithm for $d = 2$. There has been more work on the 1-RMS problem see [9, 30, 34], including a generalization by Faulkner et. al. [22] that considers non-linear utility functions.

The 1-regret problem can be easily addressed by the notion of ϵ-kernel coresets, first introduced by Agarwal et al. [1]. Later, faster algorithms were proposed to construct a coreset [10].

The 1-RMS problem is also closely related to the problem of approximating the Pareto curve (or skyline) of a set of points. Papadamitriou and Yannakakis [32, 33] considered this problem and defined an approximate Pareto curve as a set of points whose $(1 + \epsilon)$ scaling dominates every point on the skyline. They showed that there exists such a set of polynomial size [32, 33]. However, computing such a set of the smallest size is NP-Complete [23]. See also [38].

References

1. P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *Journal of the ACM (JACM)*, 51(4):606–635, 2004.
2. P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Geometric approximation via coresets. *Combinatorial and computational geometry*, 52:1–30, 2005.
3. P. K. Agarwal and M. Sharir. Arrangements and their applications. *Handbook of computational geometry*, pages 49–119, 2000.
4. A. Asudeh, A. Nazi, N. Zhang, and G. Das. Efficient computation of regret-ratio minimizing set: A compact maxima representative. In *Proc. SIGMOD*, 2017. To appear.
5. I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems (TODS)*, 33(4):31, 2008.
6. S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. 17th Int. Conf. Data Eng.*, pages 421–430, 2001.
7. H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite vc-dimension. *Discrete & Computational Geometry*, 14(4):463–479, 1995.
8. W. Cao, J. Li, H. Wang, K. Wang, R. Wang, R. Chi-Wing Wong, and W. Zhan. k-regret minimizing set: Efficient algorithms and hardness. In *ICDT 2017-20th International Conference on Database Theory*, pages 11:1–11:19, 2017.
9. I. Catallo, E. Ciceri, P. Fraternali, D. Martinenghi, and M. Tagliasacchi. Top-k diversity queries over bounded regions. *ACM Transactions on Database Systems (TODS)*, 38(2):10, 2013.

10 T. M. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 152–159. ACM, 2004.

11 D. Chen, G.-Z. Sun, and N. Z. Gong. Efficient approximate top-k query algorithm using cube index. In *Asia-Pacific Web Conference*, pages 155–167. Springer, 2011.

12 S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Computing k-regret minimizing sets. *Proceedings of the VLDB Endowment*, 7(5):389–400, 2014.

13 G. Das and M. T. Goodrich. On the complexity of optimization problems for 3-dimensional convex polyhedra and decision trees. *Computational Geometry*, 8(3):123–137, 1997.

14 G. Das, D. Gunopulos, N. Koudas, and N. Sarkas. Ad-hoc top-k query answering for data streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB'07, pages 183–194, 2007.

15 Z. Gong, G.-Z. Sun, J. Yuan, and Y. Zhong. Efficient top-k query algorithms using k-skyband partition. In *International Conference on Scalable Information Systems*, pages 288–305. Springer, 2009.

16 U. Güntzer, W. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB'00, pages 419–428, 2000.

17 Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2015. URL: `http://www.gurobi.com`.

18 J.-S. Heo, J. Cho, and K.-Y. Whang. The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 445–448. IEEE, 2010.

19 V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD Rec.*, 30(2):259–270, May 2001.

20 I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

21 S. Jasna and M. J. Pillai. An algorithm for retrieving skyline points based on user specified constraints using the skyline ordering. *International Journal of Computer Applications*, 104(11), 2014.

22 T. Kessler Faulkner, W. Brackenbury, and A. Lall. k-regret queries with nonlinear utilities. *Proceedings of the VLDB Endowment*, 8(13):2098–2109, 2015.

23 V. Koltun and C. H. Papadimitriou. Approximately dominating representatives. *Theor. Comput. Sci.*, 371(3):148–154, February 2007.

24 R. D. Kulkarni and B. F. Momin. Skyline computation for frequent queries in update intensive environment. *Journal of King Saud University-Computer and Information Sciences*, 2015.

25 R. D. Kulkarni and B. F. Momin. Parallel skyline computation for frequent queries in distributed environment. In *Computational Techniques in Information and Communication Technologies (ICCTICT), 2016 International Conference on*, pages 374–380. IEEE, 2016.

26 C. Li, Kevin K. C.-C. Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD'06, pages 61–72, 2006.

27 Q. Liu, Y. Gao, G. Chen, Q. Li, and T. Jiang. On efficient reverse k-skyband query processing. In *International Conference on Database Systems for Advanced Applications*, pages 544–559. Springer, 2012.

28 A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, June 2004.

29 M. Morse, J. M. Patel, and W. I. Grosky. Efficient continuous skyline computation. *Information Sciences*, 177(17):3411–3437, 2007.

30 D. Nanongkai, A. Lall, A. Das Sarma, and K. Makino. Interactive regret minimization. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 109–120. ACM, 2012.

31 D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu. Regret-minimizing representative databases. *Proceedings of the VLDB Endowment*, 3(1-2):1114–1124, 2010.

32 C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS'00, pages 86–92, 2000.

33 C. H. Papadimitriou and M. Yannakakis. Multiobjective query optimization. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS'01, pages 52–59, 2001.

34 P. Peng and R. C.-W. Wong. Geometry approach for k-regret query. In *2014 IEEE 30th International Conference on Data Engineering*, pages 772–783. IEEE, 2014.

35 S. Rahul and Y. Tao. Efficient top-k indexing via general reductions. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'16, pages 277–288, 2016.

36 M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the 300th International Conference on Very Large Data Bases*, VLDB'04, pages 648–659, 2004.

37 E. Tiakas, G. Valkanas, A. N. Papadopoulos, Y. Manolopoulos, and D. Gunopulos. Processing top-k dominating queries in metric spaces. *ACM Trans. Database Syst.*, 40(4):23:1–23:38, January 2016.

38 S. Vassilvitskii and M. Yannakakis. Efficiently computing succinct trade-off curves. *Theor. Comput. Sci.*, 348(2):334–356, December 2005.

39 A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 365–376. IEEE, 2010.

40 D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB'06, pages 235–246, 2006.

41 A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-k queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 397–408. ACM, 2012.

42 A. Yu, P. K. Agarwal, and J. Yang. Top-k preferences in high dimensions. *IEEE Trans. Knowl. Data Eng.*, 28(2):311–325, 2016.

43 Z. Zhang, S. w. Hwang, K. C.-C. Chang, M. Wang, C. A. Lang, and Y. c. Chang. Boolean + ranking: Querying a database by k-constrained optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD'06, pages 359–370, 2006.

A Affine transformation of polytope Π

From Lemma 3 we know that RMS problem is NP-Hard when the preferences vectors are in $\mathbb{R}^3$. Here, we show that RMS problem is NP-Hard even when preferences are restricted to $\mathbb{X}$. The reduction is similar to the reduction proposed in Lemma 3. The only difference is that polytope Π needs to have two additional properties:

(i) All vertices of Π must lie in the first orthant.

(ii) For any edge (v_1, v_2) of Π, where v_1, v_2 are vertices of P, there is a direction $u \in \mathbb{X}$ such that v_1, v_2 are the top vertices in direction u.

It is easy to satisfy property (i) because the translation of the vertices of a polytope does not change the rank of the points in any direction. On the other hand, property (ii) is not guaranteed by the construction in [13].

We show that there is an affine transformation of Π that can be computed and applied in polynomial time, to get a polytope Π' with the same combinatorial structure as Π, but that also satisfies properties (i), and (ii). The fact that the polytope has the same combinatorial structure implies that the underlying graph is the same, and therefore a vertex cover will also be a $(2, 0)$-regret set of Π. Next, we describe the details of the transformation.

Construction. First, we translate Π such that the origin o is inside Π. Then, we compute the polar dual Π^*[6]. Let v be a vertex of Π^*. Translate Π^* such that v becomes the origin. Then take a rotation such that polytope Π^* does not intersect the negative orthant – i.e., the set of points in $\mathbb{R}^3$ which have all coordinates strictly negative; we can always do it because Π^* is convex. Let u_1, u_2, u_3 be the three directions emanating from the origin such that the cone defined by them, contains the entire polytope Π^*. Such directions always exist and can be found in polynomial time. It is known that we can find in polynomial time an affine transformation such that u_1 is mapped to the direction $e_1 = (1, 0.01, 0.01)$, u_2 to direction $e_2 = (0.01, 1, 0.01)$ and u_3 to $e_3 = (0.01, 0.01, 1)$ (we can do it by first transforming u_1, u_2, u_3 to the unit axis vectors and then transform them to e_1, e_2, e_3). Apply this affine transformation to Π^* to get $\hat{\Pi}^*$. Polytope $\hat{\Pi}^*$ lies in the first orthant, except of vertex v which is at the origin. Shift this polytope slightly such that the origin lies in the interior of the polytope, v lies in the negative orthant, and all the other vertices are still in the first orthant. Such a translation can be computed in polynomial time by subtracting from all coordinates a quantity $m/2$, where m is the value of the minimum coordinate over all points except v. Hence, after the translation, $v = (-m/2, -m/2, -m/2)$. Finally we compute the polar dual of $\hat{\Pi}^*$; call this $\hat{\Pi}$. Translate $\hat{\Pi}$ until all vertices have positive coordinates, and let Π' denote the new polytope.

▶ **Lemma 11.** *Polytope Π' is combinatorially equivalent to Π and satisfies properties* (i)*,* (ii)*.*

Proof. We start by mapping property (ii) in the dual space. Consider a polytope G and its dual G^* (where the origin lies inside them). It is well known that any vertex v of G corresponds to a hyperplane h_v in the dual space that defines a facet of G^*. An edge between two vertices in G corresponds to an edge between the two corresponding faces in G^*. Furthermore, if a vertex v of G is the top-k vertex of G in a direction u, then the

[6] The polar dual of a polytope containing the origin o is defined as the intersection of all hyperplanes $\langle x, p\rangle \le 1$ where $p \in P$, and it can be equivalently defined as the intersection of the dual hyperplanes $\langle x, v\rangle \le 1$ for all the vertices v of P.

corresponding hyperplane h_v is the k-th hyperplane (among the n dual hyperplanes) that is intersected by the ray ou, where o is the origin. From the above it is straightforward to map property (ii) in the dual space: (ii') For any edge (f_1, f_2) where f_1, f_2 are faces of G^* there is a direction $u \in \mathbb{X}$ such that the first two hyperplanes that are intersected by the ray ou are h_1, h_2, where h_1 is the hyperplane that contains the face f_1 and h_2 the hyperplane that contains the face f_2.

We now show how these properties can be guaranteed in $\hat{\Pi}^*$. Notice that from the construction of $\hat{\Pi}^*$, the origin lies inside $\hat{\Pi}^*$ and all faces of $\hat{\Pi}^*$ have non empty intersection with the positive octant. By convexity, $\hat{\Pi}^*$ satisfies property (ii') because for any edge $e = (f_1, f_2)$ of $\hat{\Pi}^*$ there is a ray emanating from the origin that first intersects the edge e, and hence the hyperplanes h_1, h_2 are the first hyperplanes that are intersected by the ray. So, its dual polytope $\hat{\Pi}$ satisfies property (ii). In addition, Π^* is combinatorially equivalent to Π, by duality. Since we apply an affine transformation $\hat{\Pi}^*$ is also combinatorially equivalent to Π^*. Finally, the polytope $\hat{\Pi}$ is combinatorially equivalent to $\hat{\Pi}^*$ (its dual). Notice that translation does not change the combinatorial structure of a polytope or the ordering of the points in any direction, so Π' satisfies property (ii), property (i) by definition, and is also combinatorially equivalent to Π. ◀

The first part of the NP-Hardness proof is the same with the case of all directions in $\mathbb{R}^3$ in Lemma 3, if Q is a vertex cover of Π' then it is also a $(2, 0)$-regret set. Using property (ii) of Π', it is straightforward to show the other direction.

B Proof of Lemma 8

▶ **Lemma 8.** *Let Q' be a hitting set of Σ_{N}, and let B be the basis of P. Then $\mathsf{Q} = \mathsf{Q}' \cup B$ is a $(k, \epsilon + \delta - \delta\epsilon)$-regret set of P.*

Proof. It suffices to show that for any direction $u \in \mathbb{U}$ there is a point $q \in \mathsf{Q}$ for which $\omega(u, q) \geq (1 - \delta)(1 - \epsilon)\omega_k(u, \mathsf{P})$ (because $1 - (\epsilon + \delta - \delta\epsilon) = (1 - \delta)(1 - \epsilon)$).

We first consider the case when $\omega_k(u, \mathsf{P}) \leq \frac{1}{(1-\epsilon)\sqrt{d}}$. In this case, by the proof of Lemma 6 the set B is guaranteed to contain a point q with $\omega(u, q) \geq \frac{1}{\sqrt{d}}$, which proves the claim. So let now assume that $\omega_k(u, \mathsf{P}) > \frac{1}{(1-\epsilon)\sqrt{d}}$. Let $\bar{u} \in \mathsf{N}$ be a direction in the net N such that, $\widehat{(u, \bar{u})} \leq \delta/2d$, where $\widehat{(u, \bar{u})}$ is the angle between u and $\bar{u}$. Such a direction exists because N is a $\frac{\delta}{2d}$-net on $\mathbb{U}$. Observe that,

$$\|u - \bar{u}\| = \sqrt{2 - 2\cos(\widehat{(u, \bar{u})})} = 2\sin\left(\frac{\widehat{(u, \bar{u})}}{2}\right) \leq \frac{\delta}{2d},$$

where we have used first the cosine rule, the identity $1 - \cos\theta = 2\sin^2\left(\frac{\theta}{2}\right)$, as well as the inequality $\sin\theta \leq \theta$ for $\theta \geq 0$ in the final step. Also, observe that for any $p \in \mathsf{P}$ we have,

$$|\omega(u, p) - \omega(\bar{u}, p)| \leq \frac{\delta}{2\sqrt{d}}. \tag{1}$$

This follows because,

$$|\omega(u, p) - \omega(\bar{u}, p)| = |\langle u, p\rangle - \langle \bar{u}, p\rangle| = |\langle u - \bar{u}, p\rangle| \leq \|u - \bar{u}\| \cdot \|p\| \leq \frac{\delta}{2d} \cdot \sqrt{d} = \frac{\delta}{2\sqrt{d}},$$

where we have used the Cauchy-Schwarz inequality for the first inequality, the upper bound on $\|u - \bar{u}\|$ derived earlier, along with $\|p\| \leq \sqrt{d}$ (by Lemma 6) for the second inequality.

Let $x_1, x_2, \ldots, x_k \in \mathsf{P}$ be the top-k points along direction u, i.e., $x_i = \varphi_i(u, \mathsf{P})$. Also, let y_k be the top-k point along direction $\bar{u}$. As remarked we can assume, $\omega(u, x_i) \geq \omega(u, x_k) \geq \frac{1}{(1-\epsilon)\sqrt{d}}$. Now, for any $i = 1, 2, \ldots, k$ we have that,

$$\omega(\bar{u}, x_i) \geq \omega(u, x_i) - \frac{\delta}{2\sqrt{d}} \geq \omega(u, x_i) - \frac{(1-\epsilon)\delta}{2}\omega(u, x_i)$$
$$= \omega(u, x_i)\left(1 - \frac{(1-\epsilon)\delta}{2}\right) \geq \omega(u, x_k)\left(1 - \frac{(1-\epsilon)\delta}{2}\right).$$

The first inequality follows by *Equation* 1, and the second inequality holds since $\omega(u, x_i) \geq \omega(u, x_k) > \frac{1}{(1-\epsilon)\sqrt{d}}$. This implies that there are k points whose scores are each at least $\omega(u, x_k)\left(1 - \frac{(1-\epsilon)\delta}{2}\right)$, and therefore the k-th best score along $\bar{u}$, i.e., $\omega(\bar{u}, y_k)$, is at least $\omega(u, x_k)\left(1 - \frac{(1-\epsilon)\delta}{2}\right)$. Now, the algorithm guarantees that there is a point $q \in Q$ such that $\omega(\bar{u}, q) \geq (1-\epsilon)\omega(\bar{u}, y_k)$. We claim that this q "settles" direction u as well, up-to the factor $(1-\delta)(1-\epsilon)$. Indeed,

$$\begin{aligned}
\omega(u, q) &\geq \omega(\bar{u}, q) - \frac{\delta}{2\sqrt{d}} \geq (1-\epsilon)\omega(\bar{u}, y_k) - \frac{\delta}{2\sqrt{d}} \\
&\geq (1-\epsilon)\left(1 - \frac{(1-\epsilon)\delta}{2}\right)\omega(u, x_k) - \frac{\delta}{2\sqrt{d}} \\
&\geq (1-\epsilon)\left(1 - \frac{(1-\epsilon)\delta}{2}\right)\omega(u, x_k) - \frac{(1-\epsilon)\delta}{2}\omega(u, x_k) \\
&= (1-\epsilon)(1-\delta+\delta\epsilon/2)\omega(u, x_k) \geq (1-\delta)(1-\epsilon)\omega(u, x_k)
\end{aligned}$$

This completes the proof. ◀

Engineering an Approximation Scheme for Traveling Salesman in Planar Graphs*

Amariah Becker[1], Eli Fox-Epstein[2], Philip N. Klein[3], and David Meierfrankenfeld[4]

1 **Department of Computer Science, Brown University, Providence, RI, USA**
becker@cs.brown.edu
2 **Department of Computer Science, Brown University, Providence, RI, USA**
ef@cs.brown.edu
3 **Department of Computer Science, Brown University, Providence, RI, USA**
klein@cs.brown.edu
4 **Department of Computer Science,Brown University, Providence, RI, USA**
nfelddav@cs.brown.edu

Abstract

We present an implementation of a linear-time approximation scheme for the traveling salesman problem on planar graphs with edge weights. We observe that the theoretical algorithm involves constants that are too large for practical use. Our implementation, which is not subject to the theoretical algorithm's guarantee, can quickly find good tours in very large planar graphs.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Traveling Salesman, Approximation Schemes, Planar Graph Algorithms, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.8

1 Introduction

> "Many (if not most) polynomial-time approximation schemes (PTASs) ... for the traveling salesman problem (TSP) and related problems suffer from gigantic constant factors." –Müller-Hannemann and Schirra [17]

Overcoming gigantic constant factors. Müller-Hannemann and Schirra's introduction on algorithm engineering [17] gives this example of how the daunting complexity of algorithms and outrageous "constants," byproducts of worst-case asymptotic analysis preferred by the theory community, can lead to seemingly unimplementable algorithms. Tazari and Müller-Hannemann [21] give the first implementation of an "inherently impractical" algorithm suffering from abysmal hidden constants: a PTAS for the Steiner Tree problem on planar graphs. Their surprising outcome was a very practical implementation, one that could get reasonably good solutions on larger instances than could be addressed with previous implementations. Our work on implementation of a seemingly theoretical approximation scheme is inspired by that outcome.

The framework underlying the Steiner-tree PTAS [4] was presented in a paper [15] that illustrated the framework by presenting a linear-time approximation scheme for the Traveling

* Research funded by NSF grant CCF-14-09520. Klein's research received additional support from the Radcliffe Institute of Advanced Study, Harvard University.

© Amariah Becker, Eli Fox-Epstein, Philip N. Klein, and David Meierfrankenfeld;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 8; pp. 8:1–8:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Salesman Problem (TSP) on the metric defined by a planar graph with edge-weights. The application of the framework to this problem is perhaps the simplest illustration of the framework, though the dynamic program at its heart is harder for TSP than for Steiner tree.

Our implementation as a first step. We describe an implementation of this TSP approximation scheme, adapted and engineered so that it is no longer guaranteed to find near-optimal tours but it runs quickly. Our implementation typically runs in less than a millisecond per vertex and provides significantly better tours than similarly fast heuristics on very large graphs. Due to the magnitude of the constants involved, it was not our objective to outperform existing implementations on small graphs. Instead, this proof-of-concept implementation targets very large graphs, tests the strategies for coping with the challenges of highly theoretical algorithms outlined by Tazari and Müller-Hannemann [21], and promotes implementation within the theory community, even for the most theoretical algorithms.

Moreover, this implementation is a first step toward implementing more complicated, related approximation schemes for other problems, including *Subset TSP* (a.k.a. *Steiner TSP*), in which the tour need only visit a given subset of the vertices. It is also a first step towards implementing a method that can cope with the nonplanarities and asymmetry of real road networks. Such progress is potentially valuable because there is potential to address other problems arising in road networks, such as ride-sharing, package-delivery routing, and public transportation layout. It remains to be seen whether these techniques can be adapted to address practical applications; in this work we add to the evidence provided by Tazari and Müller-Hannemann showing that the large constant factors in the theoretical algorithms do not represent a fundamental obstacle in this effort. It is in this sense that our implementation is a proof of concept.

Alternative implementations. The literature on solving TSP exactly, approximately, and heuristically in a variety of settings is too vast to thoroughly review. Much of the literature and implementations address primarily the *Euclidean case*, in which one seeks a tour through a Euclidean space that visits all given points and has minimum length. In particular, implementations (e.g. [3, 18]) that aim for both linear time and high-quality solutions are restricted to the Euclidean case. A more general problem is *metric TSP*, in which the distances between points is a metric. One source of metrics is edge-weighted undirected graphs; the *metric completion* of such a graph is the metric space over its vertices in which the u-to-v distance is the length of a shortest u-to-v path. Note that a tour in such a metric space corresponds to a closed walk in the graph that is allowed to travel through a vertex more than once. Thus the graph need not be Hamiltonian to admit a tour.

The leading TSP codes (Concorde [1] , `LKH-2` [14], and Google's `or-tools`) require that all vertex-to-vertex distances be available, i.e. all these distances are stored in a table or there is a procedure to compute any such distance given the two vertices. This works quite well for Euclidean instances because a point-to-point distance can be computed very quickly. For instances arising from edge-weighted graphs, this is a serious obstacle: for the graphs we want to address, the number of vertices is too large for an all-pairs-distance table. There is a publicly available implementation of a high-quality distance oracle (a data structure supporting fast vertex-to-vertex distance queries) that of Dibbelt et al. [9] (`RoutingKit`), based on the algorithm of Geisberger et al. [10]. In our experiments with this implementation, however, distance-finding is not fast enough to make the existing TSP codes competitive for very large graphs. Indeed, the running time of the Lin-Kernighan heuristic is reported [13] to increase as $n^{2.2}$, and in our experience with `RoutingKit`, the time per distance was 0.04 seconds.

Moreover, the primary motivation of our work is not to beat existing codes specifically addressing TSP but to illustrate the potential of an approach to optimization problems on planar graphs and to identify algorithmic techniques that could help make this approach useful in practice.

However, it is useful to have a baseline for tour quality in order to evaluate the effectiveness of our implementation on very large graphs. There is one other theoretical algorithm that runs in linear time and finds approximately optimal tours on edge-weighted planar graphs: the 2-approximation derived from a minimum-weight spanning tree. This algorithm finds a minimum-weight spanning tree and then obtains an Euler tour of the doubled tree. We found that the minimum-spanning-tree heuristic runs very fast on large instances but produces tours that are far worse than those produced by our implementation.

One natural enhancement to the minimum-spanning-tree algorithm is to "shortcut" the tour it produces: a segment of the tour that visits only vertices already visited can be replaced by a shortest path. However, this enhancement has two drawbacks. First, carrying out the enhancement on a very large graph requires substantially more time than our implementation, even when the distance oracle of Dibbelt et al. [9] is used to compute these shortest paths. Second, the quality of the tours produced on very large graphs derived from road maps is inferior to those found by our implementation. We outline these results in Section 4.2.

What about other approximation algorithms for TSP in graphs? Christofides' algorithm [7] gives a 1.5-approximation, but requires computing a minimum-weight T-join. We are aware of no code that can compute this without first computing all-pairs distance, though Barahona [2] gives a theoretical $O(n^{1.5} \log n)$ algorithm for planar graphs.

In independent work, Xia et al. [22] reported on experiments with a graph-based method for the *Steiner TSP*, finding a tour visiting a given subset of vertices. We briefly discuss their approach in Section 1.2 since it uses a term defined in that section.

Approximating the Held-Karp lower bound. How then can we evaluate the quality of the tours obtained by our implementation? We have implemented an algorithm to compute an approximately optimal Held-Karp lower bound. The algorithm uses the packing-covering framework (see, e.g., [23], also known as multiplicative-weights update) and a dynamic algorithm for minimum-weight cuts that is based on a min-cut algorithm [5] for planar graphs. The approximate solutions we obtain for Held-Karp are enough to show that, on large graphs derived from road maps, our implementation finds tours of good quality.

Summary of our contributions. Our main contributions are as follows:

- We have implemented a theoretically linear-time approximation scheme for TSP in edge-weighted planar graphs.
- Our implementation incorporates data structures and heuristics that turn a highly theoretical algorithm into a procedure that shows promise as a practical tool.
- For appropriate settings of the parameters (going beyond what theoretical analysis allows), our implementation processes a graph in less than a millisecond per vertex and produces a tour that is empirically within 5% to 15% of optimal.
- We have also developed a procedure for deriving lower bounds on TSP in planar graphs. This enables us to measure the quality of the tours produced by our implementation.

A live demonstration of our implementation applied to graphs derived from road maps is available at `http://tsp.cs.brown.edu`.

1.1 Overview

After reviewing some fundamentals, we give a brief overview of the algorithm we implement: the linear-time approximation scheme for TSP in planar graphs [15]. We observe that setting the parameters according to theoretical analysis would lead to impractical runtimes. We show that, even with tighter analysis of the constants hidden in the running time analysis, an implementation with theoretical guarantees would require tremendous computation power. For example, achieving a 1.5-approximation might require in excess of $2^{144}n$ comparisons. Next, we describe the design choices of our implementation in detail, and discuss extensive experimental results. Finally, we compare this implementation with Tazari and Müeller-Hannemann's.

1.2 Preliminaries

We assume familiarity with TSP, graph algorithms, dynamic programming on branch decompositions, and approximation algorithms, including PTASs.

Planar graphs. Throughout, all graphs considered are planar and embedded. The *dual* of a planar graph G is the graph whose vertices are the faces of G, with edges between faces which share a boundary edge. The *radial* of a planar graph G is the bipartite graph whose vertices are the union of the vertices of G and the faces of G, with edges between each incident vertex-face pair.

Dynamic programming and branch decompositions. A *branch decomposition* of a graph is a rooted binary tree and a bijection between leaves of the tree and edges of the graph. Each edge e of the tree defines a *cluster* of graph edges, namely those edges corresponding to the tree leaves whose leaf-to-root path contains e. The *boundary* of a cluster is the set of all vertices with at least one incident edge within the cluster and one not within the cluster. The *width* of a branch decomposition is the maximum cardinality of any cluster's boundary. The *branchwidth* of a graph is the minimum width of any branch decomposition of the graph. By considering only interactions on the boundary, branch decompositions are amenable to dynamic programming.

A *sphere-cut decomposition* is a branch decomposition where, for each cluster, a Jordan curve intersects no edges and exactly the boundary vertices. This induces a cyclic order to the boundary. In a planar graph whose radial graph has radius k, a sphere-cut decomposition of width at most $k+1$ can be found in linear time using Tamaki's heuristic [20].

In independent work, Xia et al. [22] very recently reported on experiments on solving the Steiner traveling salesman problem, finding a tour that visits a given subset of vertices (*terminals*). They started with a graph derived from road maps and pruned away vertices and edges not on shortest terminal-to-terminal paths. They then found an optimal sphere-cut decomposition (i.e. one whose width is the branchwidth of the pruned graph), and then used the decomposition to find an optimal tour. The algorithm [24] for finding an optimal sphere-cut decomposition, based on the algorithm of Seymour and Thomas [19], requires $O(n^3)$ time. Given a graph and a sphere-cut decomposition of width w, the algorithm of Xia et al. takes $O(7^{2w}n^2)$ time to compute the optimal tour.

Xia et al. reported on experiments finding an optimal tour using the sphere-cut decomposition and finding an approximately optimal tour using Christofides' 1.5-approximation algorithm. However, they addressed much smaller graphs; the largest graph they considered had 300 vertices and 363 edges.

TSP. Let $\mathsf{OPT}(G)$ be the minimum cost of a TSP tour of graph G and $\mathsf{MST}(G)$ be the cost of a minimum spanning tree of G. It is well-known that $\mathsf{MST}(G) < \mathsf{OPT}(G) \leq 2\,\mathsf{MST}(G)$, giving a trivial 2-approximation algorithm. Christofides' algorithm [7] gives a 1.5-approximation, but requires computing a minimum-weight perfect matching, for which no nearly linear-time algorithm is known on planar graphs.

2 PTAS for TSP

We implement a slight variation of the Klein's PTAS [15]. In this section, we briefly summarize the algorithm. As input, we have a planar graph $G_0 = (V(G_0), E(G_0))$ equipped with an embedding and edge cost function $c(\cdot)$. We additionally have a precision parameter $\varepsilon \in (0, 1]$. The algorithm consists of four steps, described without the compromises necessary for fast runtimes:

1. **Cost reduction**: find an edge subgraph $G_1 \subseteq G_0$ of total cost at most $c(G_1) = (1 + 2/\varepsilon_1)\mathsf{MST}(G_0)$, for some constant ε_1 depending only on ε such that $\mathsf{OPT}(G_0) \leq \mathsf{OPT}(G_1) \leq (1 + \varepsilon_1)\mathsf{OPT}(G_0)$. The subroutine Spanner provided by Klein [15] satisfies these requirements and is practical.
2. **Slab decomposition**: split the graph into a collection of subgraphs called *slabs*, each of which has branchwidth at most $2/\varepsilon_2 + 3$ such that each vertex appears in at least one slab, and the sum of the costs of optimal TSP tours on the slabs is at most $\mathsf{OPT}(G_1) + 2\varepsilon_2 c(G_1)$, where ε_2 is a constant depending only on ε.
3. **Dynamic programming**: for each slab, build a branch decomposition and solve TSP exactly on it. For each cluster in the decomposition, we build a table of *configurations* and their corresponding costs: all relevant interactions between the interior and the exterior of the cluster. An upper bound on the number of configurations per cluster of width k is $M(k) = 3^{2k}$. Solving TSP exactly on a graph with a branch decomposition of width k, takes $O(3^{4k}n)$ time, since the DP performs pairwise compatibility checks of configurations from sibling clusters. (A tighter upper bound on the number of configurations per cluster is $M(k) = \sum_{i=0}^{k} C_{k-i}\binom{2k}{2i}$ where C_j is the jth Catalan number.)
4. **Combining**: return the union of the exact solutions on the slabs.

   The final output has total cost at most

$$\begin{aligned}\mathsf{OPT}(G_1) + 2\varepsilon_2 c(G_1) &\leq (1 + \varepsilon_1)\mathsf{OPT}(G_0) + 2\varepsilon_2(1 + 2/\varepsilon_1)\mathsf{MST}(G_0)\\ &< (1 + \varepsilon_1)\mathsf{OPT}(G_0) + 2\varepsilon_2\mathsf{OPT}(G_0) + 4\varepsilon_2/\varepsilon_1\mathsf{OPT}(G_0)\\ &= (1 + \varepsilon_1 + 2\varepsilon_2 + 4\varepsilon_2/\varepsilon_1)\mathsf{OPT}(G_0).\end{aligned}$$

3 Engineering Considerations

We benefit from the pessimism of worst-case analysis in several places. Before discussing how we engineer the algorithm, we note that the worst-case analysis is overly pessimistic. First, $\mathsf{MST}(G_0)$ frequently costs significantly less than $\mathsf{OPT}(G_0)$.

Second, frequently the branchwidth is less than the theoretical upper bound. Third, the total cost of slab boundaries is typically much less than $2\varepsilon_2 c(G_1)$. Furthermore, some edges in slab boundaries also belong to optimal solutions. Finally, the structure of the sphere-cut decompositions we use ensures that only rarely are two clusters merged in a way that requires considering a number of configuration pairs that is at all close to the theoretical upper limit.

Next we summarize our implementation. As part of the input, it takes parameters ε_1 and ε_2 separately, as their effects in practice differ from the theoretical guarantees.

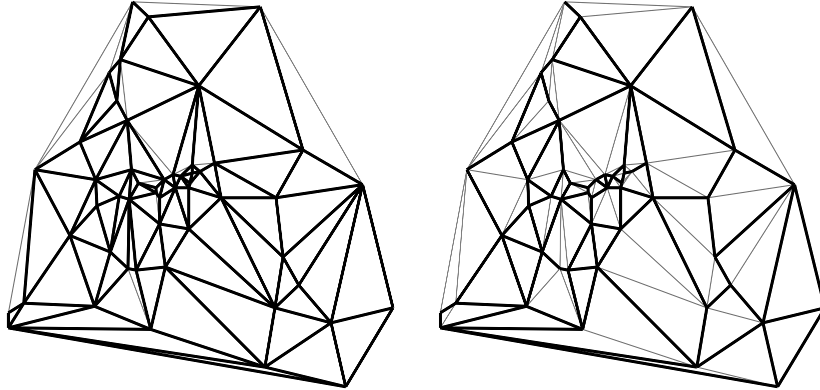


Figure 1 Delaunay triangulation of TSPLIB's `berlin52`: bold edges indicate a spanner with $\varepsilon_1 = 0.1$ (left) and $\varepsilon_1 = 0.5$ (right).

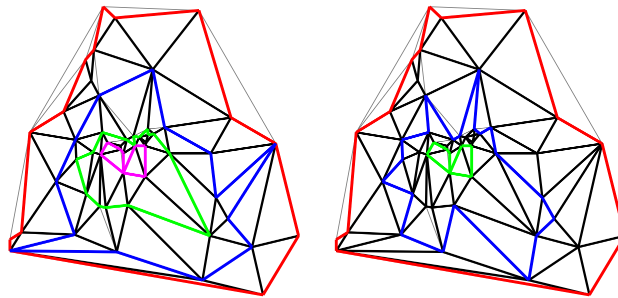
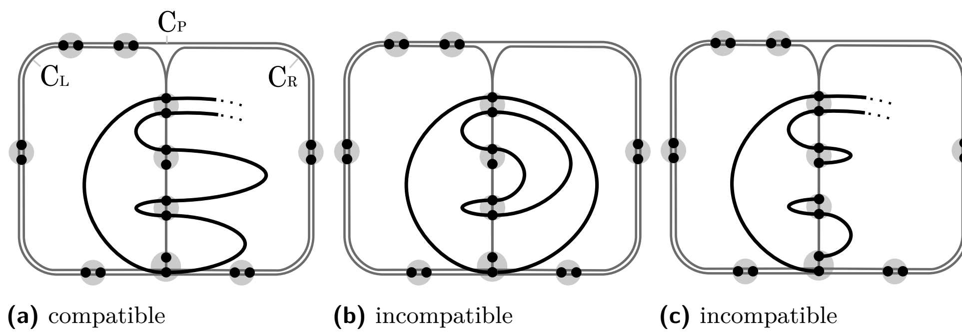


Figure 2 `berlin52`: spanner in black ($\varepsilon_1 = 0.1$) and slab boundaries for $\varepsilon_2 = 1/3$ (left) and $\varepsilon_2 = 1/4$ (right).

Cost reduction. The implementation finds graph G_1 from G_0 as in [15]. On a planar graph, this can be done in linear time [6, 16]. In practice, though, we use Kruskal's algorithm and observe that the time spent building a minimum spanning tree is usually less than 0.001% of the total runtime under reasonable choices of ε_2. Refer to Figure 1 for examples of two spanners on a small graph.

Slab decomposition. The implementation performs a breadth-first search of the dual graph. This partitions the dual edges into two types: those with endpoints on the same level of the search tree (type A) and those with endpoints on different levels (type B). Each edge is assigned a level by the minimum level of its endpoints. Level interval (i, j) consists of all type A edges with level in $[i + 1, j]$ and all type B edges with level in $[i, j]$. The type B edges of level i form the *upper seam* and the type B edges of level j form the *lower seam*. The edges in a level interval can induce a slab. The slabs used will have the property that the only edges shared between slabs are seams and the only vertices shared between slabs are incident to seam edges. Refer to Figure 2.

Dynamic program. Essentially all of the runtime is spent in the dynamic program. Because of this, most of the complexity in the implementation focuses on efficiently finding pairs of compatible configurations and merging compatible pairs into a parent configuration, and it is here that the choice of engineering techniques have the greatest impact.



(a) compatible **(b)** incompatible **(c)** incompatible

Figure 3 Prefix configuration pairs for merging child clusters C_L and C_R into parent cluster C_P: gray circles represent boundary vertices, black circles represent portals, and black lines represent tour segments. Starting with the uppermost shared vertex and going down, the left prefix configuration [(,(,),-,(,),-,)] is (a) compatible with the right prefix configuration [(,(,(,-,),(,-,)], (b) incompatible with the right prefix configuration [(,(,(,-,),),-,)] because the inner prefix cycle indicates a disconnected tour, and (c) incompatible with the right prefix configuration [(,(,(,),-,(,),-] because the crossings do not align.

Recall, each non-root, non-leaf cluster in a sphere-cut decomposition has a sibling and a parent. Furthermore, since each cluster is bounded by a Jordan curve, a natural cyclic order is assigned to the cluster's boundary vertices.

Since a TSP tour can enter or exit a cluster at most twice per vertex (subsequent crossings can be uncrossed), we split each boundary vertex into two *portals*, representing these potential connections. An involution is stored mapping portals to portals: each portal is associated with another (or to itself, when there is no entrance/exit at the portal). This involution is stored in two different ways, depending on the context: either as small integers representing the portal number or using nested parentheses (really, an array of enum objects) where matching parentheses map to one another (since TSP tours in planar graphs can be uncrossed).

The *prefix* of a cluster's portals is the interval of portals common to both children in the cyclic order induced by the cluster's bounding Jordan curve. Whether two child-cluster configurations are *compatible* can be determined mostly by comparing the section of the configurations corresponding to the prefix portals. Cycles formed between prefix portals not shared with the parent cluster indicate incompatible configurations because the final tour must be connected. Additionally, the child-cluster configurations must agree on the presence of a crossing at a prefix portal. That is, if a portal is mapped to itself on one side, it is mapped to itself on the other.[1] Refer to Figure 3 for examples of prefix compatibility.

In practice, computing the entire dynamic programming table is prohibitively expensive: merging two clusters with boundary size just 5 theoretically requires considering over 20 times more pairs of configurations than there are vertices in the largest road network publicly available for testing; in order for the algorithm to "act" like it is linear time, clusters must discard some configurations. Tazari and Müeller-Hannemann face a similar problem with large dynamic programming tables and impose a cap on the size of their dynamic programming table. We follow this strategy and limit each cluster to hold the λ best configurations found, plus one corresponding to the MST-based 2-approximation of the original graph to ensure that there is always a solution.

[1] For technical reasons, vertices common to the parent and both children pose additional complication.

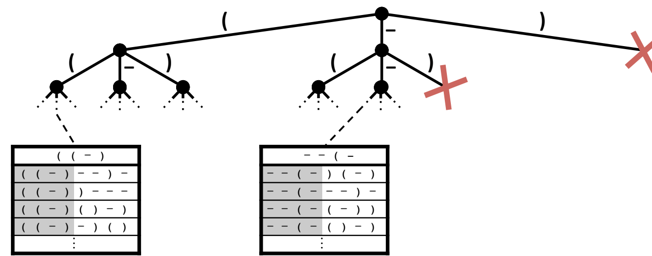


Figure 4 Each leaf of the trie corresponds to the prefix configuration generated by the root-to-leaf path. Configurations are grouped by prefix and stored (sorted by cost) at these leaves. Two such leaves are shown. The crossed-off trie branches represent invalid prefix configurations.

Rather than generating all pairs of compatible configurations and selecting the λ best, we generate them in order of increasing cost as follows. The configurations for each cluster are partitioned such that if a configuration is compatible with one configuration in a part, it is compatible with each other configuration in the part. Partitioning by equivalence classes on prefixes suffices. These parts can be efficiently stored as lists sorted by non-decreasing cost at the leaves of a trie. Each root-to-leaf path is the prefix in the nested-parenthesis representation common to all the configs stored at the leaf (refer to Figure 4). Once compatible pairs of leaves are found by traversing the tries for the child configurations in tandem, pairs of pointers to the lists' first elements are inserted into a min-heap keyed by the sum of the costs of the pointed-to elements. To get the next cheapest configuration, one pops the heap. Then, the appropriate pointer is incremented and the pair is re-inserted into the heap.

Note that just popping the heap λ times is insufficient, as some of these pairs might yield the same parent configurations. Instead, the heap is popped and parents are formed until λ have been collected. The actual formation of parent configurations from a compatible pair of child configurations is delayed until necessary, as this transformation turns out to be a bottleneck.

Post processing. As a post-processing heuristic, we draw inspiration from [8]: some number of tours, determined by an input parameter, are produced by running the full algorithm with several different slab decompositions. We then make a new graph from the union of the edges used in these tours and run the PTAS on this (unlike [8], which would just run the dynamic program on it).

Tours output by the PTAS typically are very suboptimal around slab boundaries. Recursively re-solving on the graph induced by the set of edges occurring in the tour is effective. The maximum permitted recursion depth is another parameter of the implementation.

4 Experimental Results

Our implementation consists of about 5000 lines of `C++11` with no external dependencies. We use the `g++` compiler, version 5.2.0, on the Debian 8 operating system.

We use two types of graphs for testing:

- Road networks from OpenStreetMap [12] of the major American cities of Tulsa, Dallas, Los Angeles, Rochester, and Chicago. Crossing edges are planarized by introducing a new vertex at the crossing point.
- Synthetic instances: grids with each degree-4 face randomly triangulated and random small integer costs assigned to each edge.

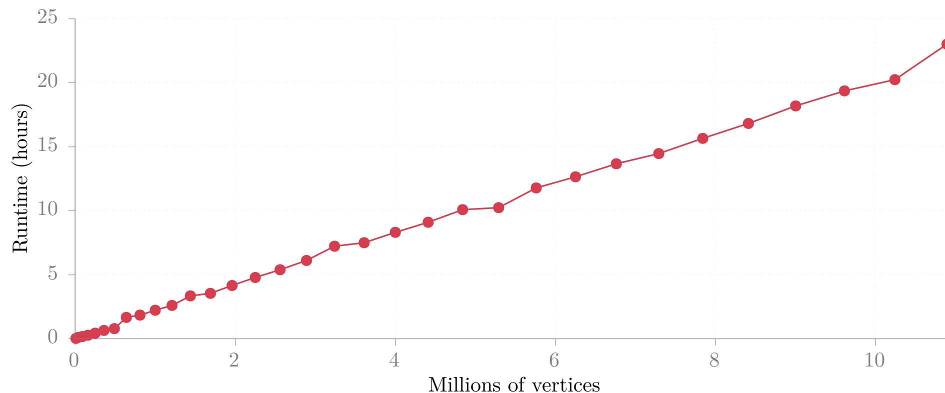


Figure 5 Runtime is linear in the size of the graph.

Table 1 Comparison of the performance of four heuristics on road networks.

Graph	# Vertices	LB	2MST		Shortcut 2MST		Fast PTAS		Slow PTAS	
			val/LB	ms/v	val/LB	ms/v	val/LB	ms/v	val/LB	ms/v
`rochester`	19488	100746068	1.41	<0.01	1.31	0.06	1.11	0.15	1.03	4.33
`tulsa`	68335	65840000	1.45	<0.01	1.34	0.22	1.11	0.23	1.04	3.41
`dallas`	403393	36332200	1.57	<0.01	1.45	2.17	1.34	0.24	1.15	4.05
`chicago`	1032016	31782700	1.49	<0.01	1.38	5.90	1.32	0.48	1.09	5.83
`losangeles`	1135323	53903389	1.44	<0.01	1.35	6.83	1.25	0.34	1.09	2.24

4.1 Linear Runtime

Our implementation exhibits linear runtime. The figure below shows the running time of the algorithm, with an arbitrary, realistic choice of parameters, on a series of synthetic square grids as described above.

Over 99.99% of the time on large instances is spent in the dynamic program; a plot of the runtime breakdown would be uninteresting.

4.2 Quality

There are two aspects of evaluating the quality of the tours returned by our algorithm: how close to optimal the tours are and how our solutions compare to other implementations. To address the former, we compute lower bounds on tour lengths, as described in the next section. For large graphs however, the latter point poses a problem. As discussed in the introduction, leading TSP implementations require all-pairs distances, which is infeasible for very large non-Euclidean instances. We compare the performance of our implementation with two different MST-based heuristics:

- The **2MST** heuristic doubles the edges of the minimum-spanning-tree.
- The **Shortcut 2MST** heuristic follows the tour of the 2MST heuristic but takes shortcuts to avoid unnecessarily re-visiting vertices.
- **Fast PTAS** is our implementation with a quicker-running set of parameters
- **Slow PTAS** is our implementation with a slower-running set of parameters

The ratios of tour lengths to lower bounds, given in Table 1 and depicted in Figure 6, provide upper bounds for solution error. We additionally report the runtime in milliseconds

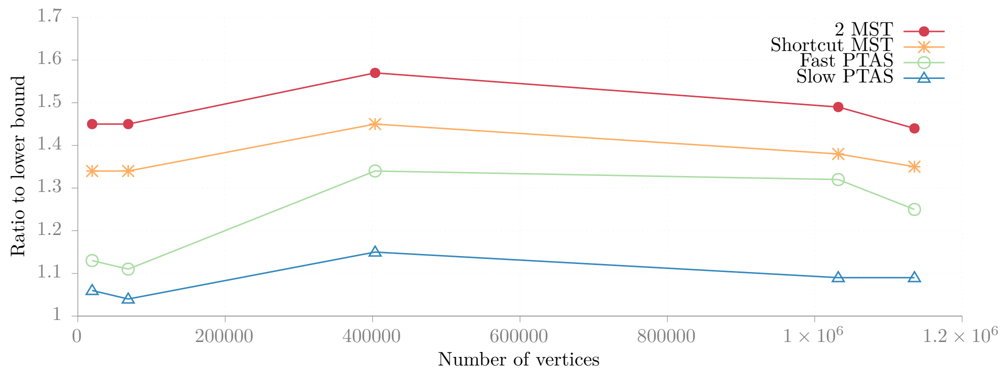


Figure 6 Visual comparison of heuristic quality on road networks.

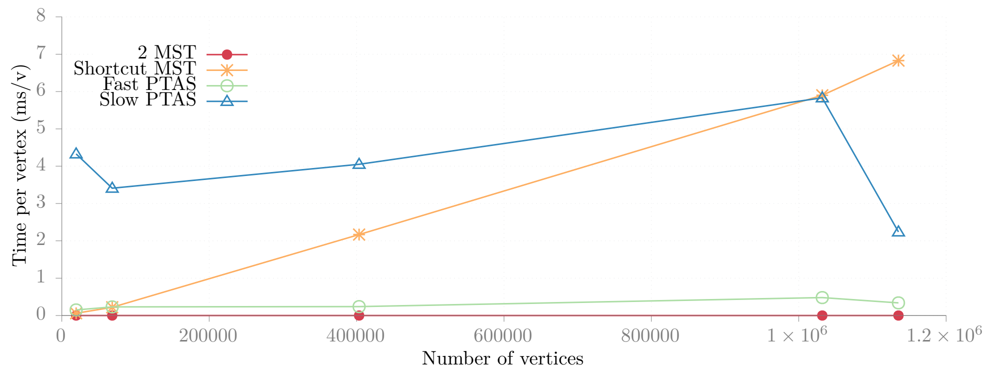


Figure 7 Visual comparison of heuristic runtime per vertex on road networks.

per vertex (refer to Figure 7). The 2MST heuristic runs extremely quickly even on very large graphs but provides a poor approximation. The Shortcut 2MST heuristic slightly outperforms the basic 2MST heuristic but takes much longer to find (the running time is superlinear in graph size). Our Fast PTAS tours are found very quickly and show a substantial improvement over 2MST, and our Slow PTAS tours are close to optimal.

4.3 The Effects of Parameters on Performance

To explore the effects of various parameters on runtime and tour cost, we ran a parameter sweep across six graphs and a variety of settings of each of four parameters: **slab height** ($1/\varepsilon_2$), number of **configurations** (λ), number of **re-solves**, and number of **tour unions**. We examined each parameter separately to identify trends in the effects on runtime and tour cost. In particular we wanted to identify parameter settings that exhibited a promising cost-runtime tradeoff. Figures 8, 9, 10, and 11 show the parameter effects on performance for the Los Angeles road network.

Figure 8 shows the effect of the **slab height** parameter. Recall that in Step 2 of the approximation scheme (Section 2, see also Section 3), the graph is decomposed into slabs. Each slab consists of a consecutive sequence of levels of a breadth-first search. The *slab height* is the number of levels comprising each slab.

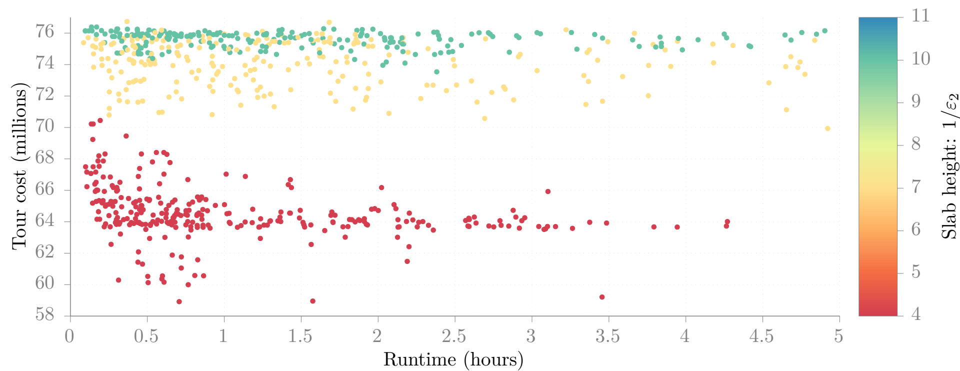


Figure 8 Performance by slab size for Los Angeles road network.

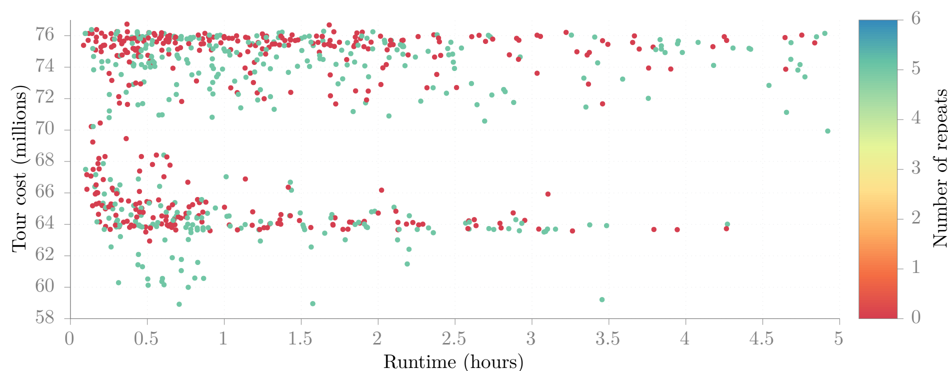


Figure 9 Performance by number of tour re-solves for Los Angeles road network.

Figures 9 and 10 shows the effect of the **re-solves** and **tour unions** parameters respectively. Recall (see *Post Processing* in Section 3) that the implementation finds several tours using different slab decompositions (all with the same height), takes the union, and repeats on the union. The number of *tour re-solves* is the number of times this happens. The *number of tour unions* refers to the size of the union taken in each repeat.

Finally, Figure 11 shows the effect of the number of **configurations** parameter. Recall that in the dynamic program, for each cluster of the branch decomposition, the implementation limits the size of the table of configurations stored for that cluster. The limit is λ, the *number of configurations*.

The number of retained configurations, λ, appears to have only a weak association to tour quality, but very fast runtimes require small λ values. Recall that the algorithm returns a tour composed of the union of slab tours which is often very suboptimal at the slab *seams*; re-solving on the graph induced by edges of the initial resulting tour (and iterating several times) can greatly improve tour quality with minimal increase in runtime. Similarly, taking the union of several solutions and re-solving on the resulting graph comprised of the union of the tours also improves tour quality. In both of these post-processing strategies the branchwidth of the graph used to re-solve TSP is typically much smaller than that of the original graph, which both substantially changes the slab decomposition and decreases the runtime.

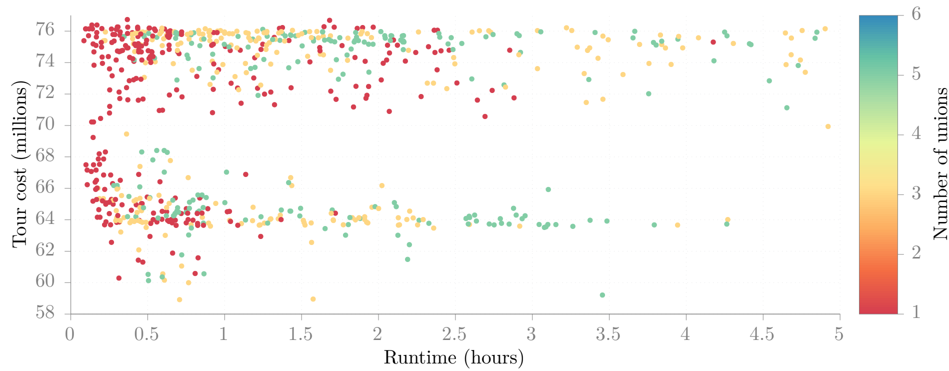


Figure 10 Performance by number of tour unions for Los Angeles road network.

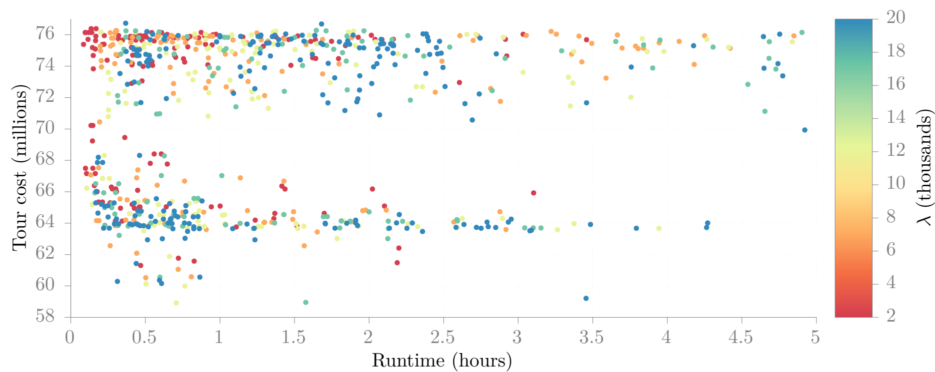


Figure 11 Performance by number of configurations for Los Angeles road network.

Interestingly and unexpectedly, larger slab heights (smaller ε_2) produce *worse* solutions. We attribute this to all values of λ being too small: the configurations kept in any cluster are only a tiny subset of potential configurations, increasing the odds of missing an important one.

Overall, we see that some parameters (such as number of repeats and unions) have clear benefits to tour quality whereas other parameters have more complicated and intricate effects and potential dependencies.

5 Computing a Lower Bound on the Traveling-Salesman Tour

We needed a way to evaluate the quality of the tours found by our code. Other implementations (e.g. Concorde and LKH) include subroutines for computing lower bounds but none supported finding a lower bound on a graph with many vertices where distances between vertices take more than a few hundred nanoseconds to compute.

5.1 The Mathematical Program

We wrote a procedure to find an approximately optimal solution to the dual of the linear program (LP) that optimizes over the subtour elimination polytope. This LP is:

$$\min\ c \cdot x\ :\ x \geq 0,\ \sum\{x_e\ :\ e \in \delta(S)\} \geq 2 \text{ for every nontrivial subset } S \subsetneq V$$

where there is a variable x_e for each edge and a constraint for each nontrivial *cut* in the graph. A nontrivial cut is the set of edges between the two parts of a bipartition of the vertices. For a subset S of vertices, $\delta(S)$ is the set of edges between S and $V - S$.

The above LP is a relaxation of TSP: any tour induces an LP solution of the same value. Therefore, the value of the LP is at most the value of the best tour.

Our procedure computes a solution to the dual of the above LP, namely

$$\max\ \vec{2} \cdot y\ :\ y \geq 0,\ \sum\{y_S\ :\ e \in \delta(S)\} \leq c_e \text{ for every edge } e.$$

This LP has a variable y_S assigning a weight to every cut $\delta(S)$. For each edge e, the total weight of cuts containing e is required to be at most the cost of e. The goal is to maximize twice the sum of the cut edges. This is called a *packing of cuts*. By LP duality, the value of this LP equals the value of the LP with the subtour elimination constraints.

5.2 Approximation Scheme

Our procedure approximates the value of the packing LP using an approximation scheme of Young [23] for solving fairly general mathematical programs (*packing/covering*) via solving a sequence of simpler mathematical programs. In this application of the method, in each iteration the procedure must find a cut whose weight is less than a threshold. The weights are adjusted in each iteration. In particular, in each iteration the procedure increases the weights of edges in the cut just selected, and adjusts the threshold. The number of iterations grows as $O(\epsilon^{-2} m \log m)$ where m is the number of edges. Each iteration of the implementation takes a step that is larger than that prescribed by theory; the implementation uses binary search to find the largest step size that preserves the algorithm's invariant.

The main work in each iteration is to find a cut of weight less than the threshold. There is a near-linear-time algorithm [5] for min-weight cut in planar graphs. The algorithm uses shortest-path separators, divide-and-conquer, and an $O(n \log n)$ algorithm for minimum *st*-cut in a planar graph. We implemented this algorithm but using it to implement an iteration is far too slow for our purposes. We therefore used it as the basis for a dynamic min-cut algorithm.

5.3 Dynamic Algorithm for Min-Weight Cut in Planar Graphs

The divide-and-conquer algorithm forms a balanced binary tree, a recursive-decomposition tree: each internal node has an associated min *st*-cut instance on a subgraph, and each leaf has an associated global min-cut instance. The dynamic algorithm maintains a priority queue of solutions to these instances, ordered according to the weights of the solutions. However, the algorithm does not automatically update the solutions or the priority queue when edge-weights increase.

When the LP algorithm requests a cut of weight less than a threshold, the dynamic algorithm examines the cut in the priority queue whose key is smallest, and computes the true weight of the cut (i.e. with respect to current edge-weights). If the true weight is less than the threshold, the dynamic algorithm returns it; if not, the algorithm puts the

corresponding instance in a queue of instances to reprocess, and moves on to the next cut in the priority queue. Once the cuts in the priority queue are exhausted and no cut of weight less than the threshold has been found, the algorithm turns to the queue of instances to reprocess; it selects the smallest of these instances and recomputes the corresponding cut. If that cut's weight is still not less than the threshold, the algorithm goes to the next larger instance, and so on. If this queue is exhausted, the algorithm starts from scratch, recomputing shortest-path separators and the recursive-decomposition tree.

5.4 Experiments

As predicted by theory, the runtime of the lower bound procedure depends quadratically on the inverse of the precision parameter ϵ. Also as predicted by theory, the number of iterations grows as $O(n \log n)$. The runtime appears to scale slightly superlinearly with the size of the graph, illustrating the empirical effectiveness of our dynamic min-cut algorithm.

The data are shown in the Appendix A.

6 Discussion

We implemented a PTAS for TSP on planar graphs designed to handle instances with millions of vertices in a reasonable amount of time. Our analysis demonstrates that, despite the significant hurdles presented by massive constants obscured in asymptotic notation, with a bit of engineering, highly theoretical algorithms can become practical. Data used for the road-map experiments will be made available at `http://tsp.cs.brown.edu`.

Comparison with Steiner tree. Like [21], we found that the maximum table size, λ, has a major effect on runtime and quality. Implementing heuristics for pruning tables more effectively seems like a viable strategy toward engineering a better, faster implementation.

In the "Conclusion and Outlook" section [21], Tazari and Müller-Hannemann write that it "would be very interesting to see how the PTAS performs for [TSP] and especially, [sic] if one can drastically reduce the required table sizes using well-known lower and upper bounds for the TSP." Unfortunately, it seems that the additional complexity of configurations in the dynamic program for TSP (compared to those in Steiner tree) means that tables are *not* able to be significantly smaller.

In Steiner tree, the configuration for a cluster boundary can be represented completely by a non-crossing partition of the vertices of the boundary; this bounds the number of configurations for a cluster of width k at C_k, which is significantly smaller than the $\sum_{i=0}^{k} C_{k-i}\binom{2k}{2i}$ we encounter. For example, with a boundary size of 5, there are at most 42 Steiner tree configurations and over 2,000 TSP configurations. Furthermore, actually combining configurations or determining if they are compatible is significantly more complicated.

Potential improvements. Our tours on the larger graphs might be better relative to the optimum than we have reported: running the lower-bound code for longer will improve those lower bounds.

The dynamic program can be easily parallelized. Considerable speedup might therefore be achieved on multi-core processors by using parallel processing.

Racing several techniques to solve the slabs and taking whichever finishes first could provide a significant speedup; although the slabs can contain $O(n)$ vertices, frequently they are small and could be solved more quickly with other techniques, e.g. local search. Simply post-processing the tour using local search is likely to significantly improve the tour length.

Upper and lower bounds could be used to prune away entire subtrees of the trie storing configurations. A deeper understanding of which branch decompositions lead to good solutions given a small λ value might improve quality significantly. Additionally, different algorithms may yield lower-width branch decompositions; experimenting with these may be fruitful.

Integrating our lower-bound computations into the dynamic program could prove fruitful by determining which clusters have near-optimal-cost configurations and which need more time invested. (Currently the lower-bound code is too slow to serve in that way.)

Tamaki's heuristic [20] can be generalized [11] to handle planar hypergraphs. It might be possible to use this to handle the localized nonplanarities that arise in road maps.

Adaptation to related problems. As discussed earlier, this is a first step in providing a robust implementation capable of handling a suite of related problems, such as Steiner TSP. To adapt to these other problems, one might need a more involved spanner step, as well as to modify the configuration compatibility-checking code. Most of the engineering techniques applied here will translate, often without modification, to related problems.

References

1. D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde TSP solver, 2006.
2. F. Barahona. Planar multicommodity flows, max cut, and the chinese postman problem. In *Polyhedral Combinatorics, Proceedings of a DIMACS Workshop, Morristown, New Jersey, USA, June 12-16, 1989*, pages 189–202, 1990.
3. J. J. Bartholdi and L. K. Platzman. Heuristics based on spacefilling curves for combinatorial problems in euclidean space. *Management Science*, 34(3):291–305, 1988.
4. G. Borradaile, P. N. Klein, and C. Mathieu. An $O(n \log n)$ approximation scheme for steiner tree in planar graphs. *ACM Trans. Algorithms*, 5(3):31:1–31:31, 2009.
5. P. Chalermsook, J. Fakcharoenphol, and D. Nanongkai. A deterministic near-linear time algorithm for finding minimum cuts in planar graphs. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 828–829, 2004.
6. D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5(4):724–742, 1976.
7. N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon University, 1976.
8. W. Cook and P. Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.
9. J. Dibbelt, B. Strasser, and D. Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, 2016.
10. R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.
11. Q. P. Gu and H. Tamaki. Improved bounds on the planar branchwidth with respect to the largest grid minor size. *Algorithmica*, 64(3):416–453, 2012.
12. M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.
13. K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.
14. K. Helsgaun. General k-opt submoves for the Lin–Kernighan TSP heuristic. *Mathematical Programming Computation*, 1(2-3):119–163, 2009.

15 P. N. Klein. A linear-time approximation scheme for TSP in undirected planar graphs with edge-weights. *SIAM Journal on Computing*, 37(6):1926–1952, 2008.

16 T. Matsui. The minimum spanning tree problem on a planar graph. *Discrete Applied Mathematics*, 58(1):91–94, 1995.

17 M. Müller-Hannemann and S. Schirra, editors. *Algorithm engineering: bridging the gap between algorithm theory and practice*, volume LNCS 5971. Springer, 2010.

18 G. Reinelt. Fast heuristics for large geometric traveling salesman problems. *ORSA Journal on Computing*, 4(3):206–217, 199.

19 P. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

20 H. Tamaki. A linear time heuristic for the branch-decomposition of planar graphs. In *Algorithms – ESA 2003*, volume 2832 of *Lecture Notes in Computer Science*, pages 765–775. Springer, 2003.

21 S. Tazari and M. Müller-Hannemann. Dealing with large hidden constants: Engineering a planar Steiner tree PTAS. *Journal of Experimental Algorithmics (JEA)*, 16:3–6, 2011.

22 Y. Xia, M. Zhu, Q. Gu, L. Zhang, and X. Li. Toward solving the steiner travelling salesman problem on urban road maps using the branch decomposition of graphs. *Information Sciences*, 374:164–178, 2016.

23 N. E. Young. Sequential and parallel algorithms for mixed packing and covering. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 538–546, 2001.

24 M. Zhu. Computational study on branch decompositions of planar graphs. Master's thesis, School of Computing Science, Simon Fraser University, 2013.

A Experiments with lower-bound procedure

As predicted by theory, the number of iterations grows as $O(n \log n)$. Here the curve is $50\, n\, \log n$:

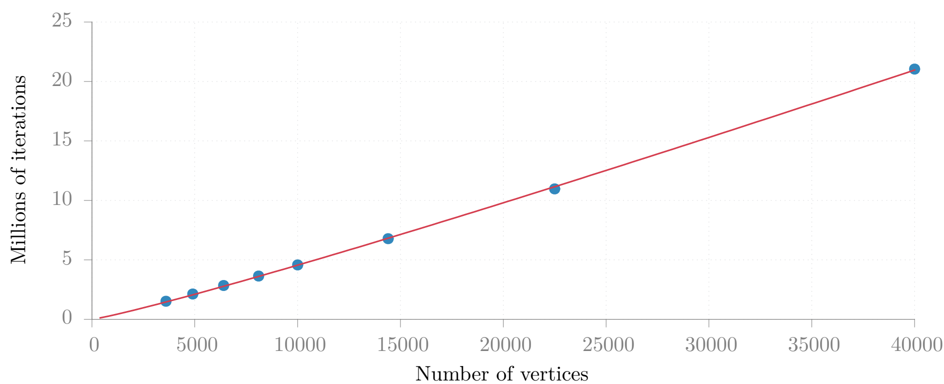


Figure 12 Lower-bound code: Number of interations grows as $50\, n\, \log\, n$ for graph size n.

The runtime in minutes of several runs with different values of ε on a road network of Rochester, NY is illustrated, along with the curve $0.008/\varepsilon^2 - 0.18$.

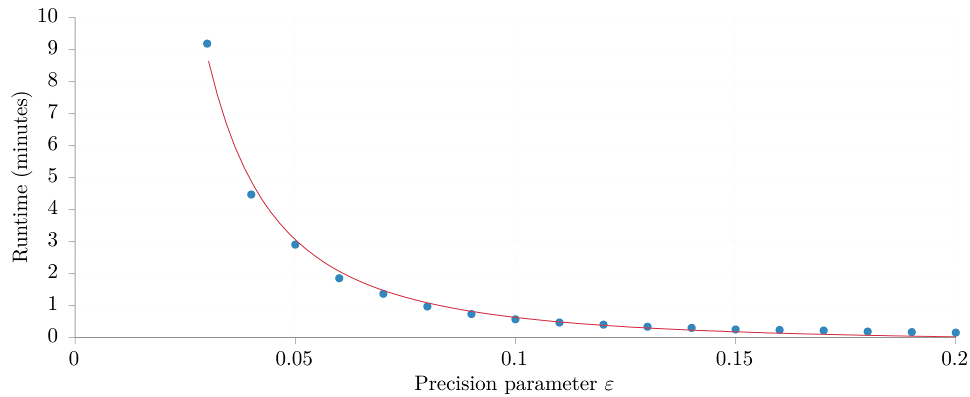


Figure 13 Lower-bound code: Runtime shrinks as $0.008/\varepsilon^2 - 0.18$ for precision parameter ε.

As visualized below, the runtime of the lower bound code appears to scale slightly superlinearly with the size of the graph. This shows the empirical effectiveness of our dynamic-min-cut algorithm. This plot shows runtime on a variety of synthetic grids with $\varepsilon = 0.05$.

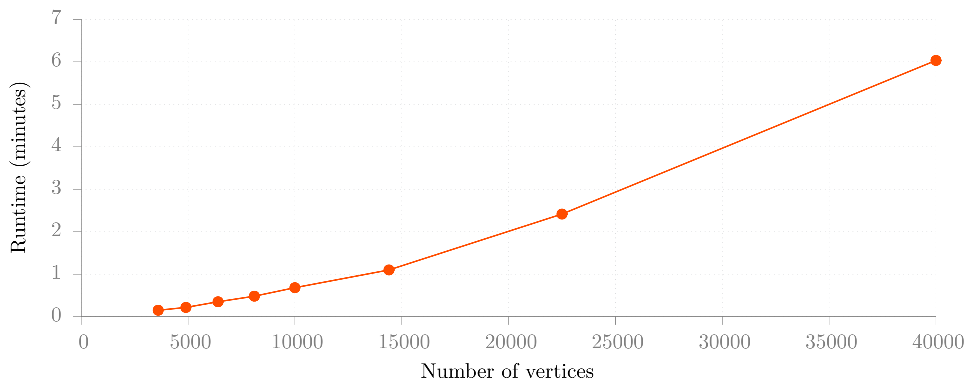


Figure 14 Lower-bound code: Runtime as a function of graph size.

Approximating the Smallest 2-Vertex-Connected Spanning Subgraph via Low-High Orders

Loukas Georgiadis[1], Giuseppe F. Italiano[2], and Aikaterini Karanasiou[3]

1 **University of Ioannina, Ioannina, Greece**
loukas@cs.uoi.gr
2 **University of Rome Tor Vergata, Rome, Italy**
giuseppe.italiano@uniroma2.it
3 **University of Rome Tor Vergata, Rome, Italy**
Aikaterini.Karanasiou.@uniroma2.it

Abstract

Let $G = (V, E)$ be a 2-vertex-connected directed graph with m edges and n vertices. We consider the problem of approximating the smallest 2-vertex connected spanning subgraph (2VCSS) of G, and provide new efficient algorithms for this problem based on a clever use of low-high orders. The best previously known algorithms were able to compute a 3/2-approximation in $O(m\sqrt{n}+n^2)$ time, or a 3-approximation faster in linear time. In this paper, we present a linear-time algorithm that achieves a better approximation ratio of 2, and another algorithm that matches the previous 3/2-approximation in $O(m\sqrt{n} + n^2)$ time. We conducted a thorough experimental evaluation of all the above algorithms on a variety of input graphs. The experimental results show that both our two new algorithms perform well in practice. In particular, in our experiments the new 3/2-approximation algorithm was always faster than the previous 3/2-approximation algorithm, while their two approximation ratios were close. On the other side, our new linear-time algorithm yielded consistently better approximation ratios than the previously known linear-time algorithm, at the price of a small overhead in the running time.

1998 ACM Subject Classification E.1 [Data Structures] Graphs and Networks, Trees, G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases 2-vertex connectivity, approximation algorithms, directed graphs

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.9

1 Introduction

The problem of approximating subgraphs that satisfy certain connectivity requirements has received a lot of attention (see, e.g., [9], and the survey [21]). In general, computing efficiently small spanning subgraphs that retain some desirable properties of an input graph is of particular importance when dealing with large-scale networks (e.g., networks with hundreds of million to billion edges), which arise often in today's applications. In this framework, designing practically efficient algorithms is also of the utmost importance. In particular, one of the biggest challenge is to design fast linear-time algorithms, since algorithms with higher running times might be practically infeasible on large-scale networks.

Before defining formally our problems, we need some preliminary definitions. Let $G = (V, E)$ be a strongly connected directed graph (digraph) with m edges and n vertices. A vertex x of G is a *strong articulation point* if $G \setminus x$ is not strongly connected, i.e., the removal of x destroys the strong connectivity of G. A strongly connected digraph G is *2-vertex-connected*

© Loukas Georgiadis, Giuseppe F. Italiano, and Aikaterini Karanasiou;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 9; pp. 9:1–9:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

if it has at least three vertices and no strong articulation points. More generally, a strongly connected digraph is *k-vertex connected* if it has at least $k+1$ vertices and the removal of any set of at most $k-1$ vertices leaves the graph strongly connected. The computation of a smallest (i.e., with minimum number of edges) k-vertex-connected spanning subgraph (kVCSS) of a given k-vertex-connected graph is a fundamental problem in network design.

In this paper, we consider the problem of approximating the smallest 2-vertex connected spanning subgraph (2VCSS) of a 2-vertex connected digraph G. The current best approximation ratio for this problem is 3/2, and it was achieved first by the algorithm by Cheriyan and Thurimella [5], which runs in $O(m^2)$ time. Georgiadis [11] presented a faster linear-time algorithm which achieves a 3-approximation. He then combined his algorithm with the 3/2-approximation algorithm of Cheriyan and Thurimella [5] to achieve a new 3/2-approximation algorithm which runs in faster $O(m\sqrt{n}+n^2)$ time. As explicitly mentioned in [11], the previous experimental study on approximation algorithms for the 2VCSS problem by Georgiadis [11] focused mainly on the solution quality achieved in practice, and not much effort was put into optimizing the running time of the algorithms considered.

The main contributions of this paper are two new efficient algorithms for this problem which exploit in a novel fashion the low-high order of a digraph [14]. Specifically, we first provide a linear-time algorithm that achieves a better approximation ratio of 2, thus improving significantly the best previous approximation ratio achievable in linear time for this problem [11]. Next, we show how to combine our new linear-time algorithm with the 3/2-approximation algorithms of Cheriyan and Thurimella [5] for 2VCSS and of Zhao et al. [27] for approximating the smallest strongly connected spanning subgraph (SCSS), so as to obtain an algorithm that achieves a 3/2-approximation in $O(m\sqrt{n}+n^2)$ time for 2VCSS. Hence, our new algorithm matches the previously known best bounds of [11].

To assess their practical value, we conducted a thorough experimental evaluation of all the above algorithms on a variety of input graphs. In order to make a fair comparison, in addition to the efficient implementations of our new algorithms, we also provide newly engineered and faster implementations of the algorithms by Georgiadis [11], which have better running times in practice while still achieving the same approximation ratios.

Our experimental results show that both our two new algorithms perform well in practice. In particular, in our experiments the new 3/2-approximation algorithm kept essentially the same approximation ratio as the previous algorithm, but it was significantly faster. On the other side, our new linear-time algorithm yielded consistently better approximation ratios than the previously known linear-time algorithm, at the price of a small overhead in the running time.

We observe that recent work [12, 13] considered also slightly more general problems than the one considered in this paper, such as approximating the smallest strongly connected spanning subgraph that maintains 2-connectivity relations of a strongly connected digraph G (where G is not necessarily 2-vertex-connected). Some of the results in this paper extend directly to this setting as well. For instance, our new linear-time 2-approximation algorithm for 2VCSS immediately implies a linear-time 2-approximation algorithm for computing the smallest strongly connected spanning subgraph of G that maintains the maximal 2-vertex-connected subgraphs of G.

2 Preliminaries

In this section, we review some basic notions and results used in our algorithms. A *flow graph* $G=(V,E,s)$ is a directed graph (digraph) with a distinguished start vertex $s \in V$

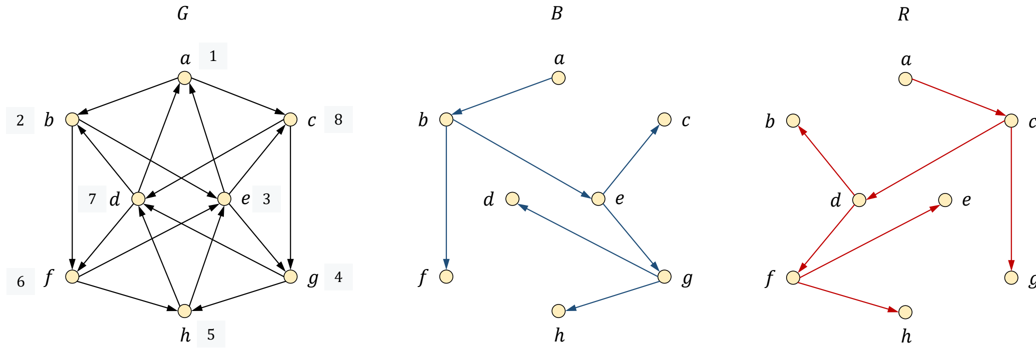


Figure 1 A 2-vertex-connected digraph G with vertices numbered in a low-high order (left); two divergent spanning trees B and R of G rooted at vertex a (right).

such that all vertices are reachable from s. The *dominator relation* in G is defined as follows. A vertex v is a *dominator* of a vertex w (v *dominates* w) if every path from s to w contains v; v is a *proper dominator* of w if v dominates w and $v \neq w$. The dominator relation in G can be represented by a tree rooted at s, the *dominator tree* D, such that v dominates w if and only if v is an ancestor of w in D. Throughout the paper, for each vertex $v \neq s$ we let $d(v)$ denote the parent of v in D. The dominator tree is a central tool in program optimization and code generation [6], and it has applications in other diverse areas [16]. The dominator tree of a flow graph can be computed in linear time [1, 4].

A spanning tree T of a flow graph G is a tree with root s that contains a path from s to v for all vertices v.

Given a rooted tree T, we denote by $T(v)$ the subtree of T rooted at v (we also view $T(v)$ as the set of descendants of v).

Let T be a tree rooted at s with vertex set V, and let $t(v)$ denote the parent of a vertex v in T. If v is an ancestor of w, we denote by $T[v, w]$ the path from v to w in T. In particular, $D[s, v]$ consists of the vertices that dominate v. If v is a proper ancestor of w, $T(v, w]$ is the path to w from the child of v that is an ancestor of w. A tree T is *flat* if its root is the parent of every other vertex.

A *preorder* of T is a total order of the vertices of T such that, for every vertex v, the descendants of v are ordered consecutively, with v first.

A *low-high order* δ *of* G [14] is a preorder of the dominator tree D with the following property: for all vertices $v \neq s$, either $(d(v), v) \in E$ or there are two edges $(u, v) \in E$, $(w, v) \in E$ such that u is less than v ($u <_\delta v$), v is less than w ($v <_\delta w$), and w is not a descendant of v in D. . Note that if D is flat, then the above definition of a low-high order δ is simplified as follows: for all vertices $v \neq s$, either $(s, v) \in E$ or there are two edges $(u, v) \in E$, $(w, v) \in E$ such that $u <_\delta v$ and $v <_\delta w$. See Figure 1. Every flow graph G has a low-high order, computable in linear-time [14]. Low-high orders provide a correctness certificate for dominator trees that is straightforward to verify [26], and also have applications in path-determination problems [14, 25] and in fault-tolerant network design [2, 3, 15].

Let $G = (V, E, s)$ be a flow graph, and let D be a dominator tree of G. Fix a low-high order δ of G and let $E' \subseteq E$ be a subset of edges of G. We say that E' *satisfies* δ if for any vertex $v \neq s$ we have that either $(d(v), v) \in E'$ or there are two edges $(u, v) \in E'$, $(w, v) \in E'$ such that $u <_\delta v$ and $v <_\delta w$, and w is not a descendant of v in D. If $E' \subseteq E$ satisfies δ, then $G' = (V, E', s)$ is a flow graph with the same dominator tree as G.

A notion closely related to low-high orders is that of divergent spanning trees [14].

Let $G = (V, E, s)$ be a flow graph. Two spanning trees B and R of G, rooted at s, are *divergent* if for all v, the paths from s to v in B and R share only the dominators of v, i.e., $B[s, v] \cap R[s, v] = D[s, v]$.

Every flow graph has a pair of divergent spanning trees. Given a low-high order of G, two divergent spanning trees of G can be computed in time $O(m + n)$ [14].

Let $G = (V, E)$ be a strongly connected graph. Note that, for any arbitrarily selected start vertex s in V, $G_s = (V, E, s)$ is a flow graph. Since there is no danger of ambiguity, in the following we will denote by G both the original strongly connected graph G and the flow graph G_s. We denote by $G^R = (V, E^R)$ the *reverse digraph* of G that results from G after reversing all edge directions. Arbitrarily fix a start vertex s in V: similarly to before, we also denote by G^R the flow graph with start vertex s, and by D^R the dominator tree of the flow graph G^R. As proved in [19], a vertex $v \neq s$ is a strong articulation point of G if and only if v is not a leaf in D or not a leaf in D^R. This implies the following property, which will be used throughout the paper:

▶ **Property 1.** *A strongly connected graph G is 2-vertex-connected if and only if:*
(a) *Both D and D^R are flat, and*
(b) *$G \setminus s$ is strongly connected.*

3 A linear-time 2-approximation algorithm

In this section, we present our new linear-time algorithm that computes a 2-approximation to the smallest 2-vertex-connected subgraph (2VCSS) of a 2-vertex-connected digraph G. The algorithm, which we call LH-Z, exploits the properties of low-high orders and uses the algorithm of Zhao et al. [27] for computing approximate smallest strongly connected spanning subgraphs (SCSS). LH-Z, described in Algorithm 1 as pseudocode, works as follows. We first choose arbitrarily a vertex s in G and start with an approximate smallest strongly connected spanning subgraph H of $G \setminus s$, which can be computed with the algorithm of Zhao et al. [27] (lines 1–3). We then compute a low-high order of the flow graph G with start vertex s (line 4); next, we add edges to H so as to ensure that the edge set of H satisfies δ, that is, δ is also a low-high order for all vertices $v \neq s$ in H (lines 5–17). This step is repeated also for the reverse flow graph G^R, with the same start vertex s (line 18). We start by proving that the spanning subgraph computed by Algorithm LH-Z is 2-vertex-connected.

▶ **Lemma 2.** *Algorithm LH-Z computes a 2-vertex-connected spanning subgraph of G.*

Proof. Let H be the subgraph computed by LH-Z. To show that H is 2-vertex connected, we prove that Property 1 holds. Note that (b) trivially holds because H is initially a strongly connected spanning subgraph of $G \setminus s$ (line 2), and it remains so after adding edges. It thus remains to show that both H and H^R have flat dominator trees. We only prove this for H, since the same argument applies to H^R.

Let δ be the low-high order of G computed in line 4. We argue that after the execution of the for loop in lines 5–17, δ must be also a low-high order in H (i.e., the edges of H satisfy δ). Consider an arbitrary vertex $v \neq s$. Let (x, v) be an edge entering v in the initial strongly connected spanning subgraph of G computed in line 2. If $x >_\delta v$, then, by the definition of δ, there is at least one edge $(y, v) \in E$ such that $y <_\delta v$. (Note that we can have $y = s$ since $s <_\delta v$ for all $v \neq s$.) Hence, after the execution of the for loop for v, the edge set E_H will contain at least two edges (u, v) and (w, v) such that $u <_\delta v <_\delta w$. On the other hand, if $x <_\delta v$, then the definition of δ implies that there is an edge $(y, v) \in E$ such that $y >_\delta v$ or $y = s$. Notice that in either case $y \neq x$ because (x, v) is an edge of $G \setminus s$. So, again, after the

---

Algorithm 1: LH-Z(G)

Input: 2-vertex-connected digraph $G = (V, E)$
Output: 2-approximation of a smallest 2-vertex-connected spanning subgraph $H = (V, E_H)$ of G

1. Choose an arbitrary vertex s of G as start vertex.
2. Compute a strongly connected spanning subgraph $H = (V \setminus s, E_H)$ of $G \setminus s$.
3. Set $H \leftarrow (V, E_H)$.
4. Compute a low-high order δ of flow graph G with start vertex s.
5. **foreach** *vertex* $v \neq s$ **do**
6.   **if** *there are two edges* (u, v) *and* (w, v) *in* E_H *such that* $u <_\delta v$ *and* $v <_\delta w$ **then**
7.   do nothing
8.   **end**
9.   **else if** *there is no edge* $(u, v) \in E_H$ *such that* $u <_\delta v$ **then**
10.   find an edge $e = (u, v) \in E$ with $u <_\delta v$
11.   set $E_H \leftarrow E_H \cup \{e\}$
12.   **end**
13.   **else if** *there is no edge* $(w, v) \in E_H$ *such that* $v <_\delta w$ **then**
14.   find an edge $e = (w, v) \in E$ with $v <_\delta w$ or $w = s$
15.   set $E_H \leftarrow E_H \cup \{e\}$
16.   **end**
17. **end**
18. Execute the analogous steps of lines 4–17 for the reverse flow graph G^R with start vertex s.
19. **return** $H = (V, E_H)$

---

execution of the for loop for v, the edge set E_H will contain at least two edges (u, v) and (w, v) such that either $u <_\delta v <_\delta w$, or $u <_\delta v$ and $w = s$. It follows that δ is a low-high order for all vertices $v \neq s$ in H.

As proved in [14], this implies that H contains two divergent spanning trees B and R of G. Since G is 2-vertex-connected, it has a flat dominator tree, and thus we have that $B[s, v] \cap R[s, v] = \{s, v\}$ for all $v \in V \setminus s$. Hence, since H contains B and R, the dominator tree of H is also flat. ◀

We remark that the construction of H in algorithm LH-Z guarantees that s will have in-degree and out-degree at least 2 in H. (This fact is implicit in the proof of Lemma 2.) Indeed, H will contain the edges from s to the vertices in $V \setminus s$ with minimum and maximum order in δ, and the edges entering s from the vertices in $V \setminus s$ with minimum and maximum order in δ^R.

▶ **Theorem 3.** *Algorithm LH-Z computes a 2-approximation for 2VCSS in linear time.*

Proof. We first establish the approximation ratio of LH-Z by showing that $|E_H| \leq 4n$. The approximation ratio of 2 follows from the fact that any vertex in a 2-vertex-connected digraph must have in-degree at least two. In line 2 we can compute an approximate smallest strongly connected spanning subgraph H of $G \setminus s$ [20]. For this, we can use the linear-time algorithm of Zhao et al. [27], which selects at most $2(n-1)$ edges. Now consider the edges selected in the for loop of lines 5–17. Since after line 2, $H \setminus s$ is strongly connected, each vertex $v \in V \setminus s$ has at least one entering edge (x, v). If $x <_\delta v$ then lines 10–11 will not be executed;

otherwise, $v <_\delta x$ and lines 14–15 will not be executed. Thus, the for loop of lines 5–17 adds at most one edge entering each vertex $v \neq s$. The same argument implies that the analogous steps executed for G^R add at most one edge leaving each vertex $v \neq s$. Hence, at the end of the execution E_H contains at most $4(n-1)$ edges.

Note that the algorithm by Zhao et al. [27] runs in linear time, and a low-high order can also be computed in linear-time [14]. Furthermore, all other steps of Algorithm LH-Z can be implemented in linear time. This yields the lemma. ◀

▶ Remark. In line 2 of algorithm LH-Z, we can alternatively set H to be the union of two spanning trees as follows. We choose an arbitrary vertex $s' \neq s$ as the start vertex of $G \setminus s$ and compute two spanning trees T and T^R of the flow graphs $G \setminus s$ and $(G \setminus s)^R$, respectively, rooted at s'. Then, we let H consist of the edges $\{(u,v) : (u,v) \in T\} \cup \{(u,v) : (v,u) \in T^R\}$, which are at most $2(n-1)$ as required by the proof of Theorem 3. In our implementation, however, we use the algorithm of Zhao et al. [27] instead. This way, we obtained better results in practice.

4 A 3/2-approximation algorithm

In this section we present a new algorithm, called LH-Z-CT, that combines our linear-time algorithm LH-Z described in Section 3 with the 3/2-approximation algorithm of Cheriyan and Thurimella [5]. We first describe a simple filtering algorithm that computes a minimal 2VCSS, and then give an overview of the Cheriyan-Thurimella algorithm.

Let $G = (V, E)$ be the input 2-vertex-connected digraph. A simple $O(m^2)$-time algorithm that gives a 2-approximation $G' = (V, E')$ of the smallest 2VCSS of G filters out redundant edges as follows: Initially, we set $G' = G$. Then, we process the edges of E in an arbitrary order: when we process an edge (x, y) we test if $G' \setminus (x, y)$ contains at least two vertex-disjoint paths from x to y. If this is the case, then we remove the edge (x, y) from E'; otherwise, we keep the edge (x, y) in E' and proceed with the next edge. Clearly, at the end of this procedure G' is a minimal 2VCSS of G, i.e., for any edge $(x, y) \in E'$, $G' \setminus (x, y)$ is not 2-vertex-connected. We refer to this algorithm as MINIMAL.

Testing if a digraph G has two vertex-disjoint paths from x to y can be done in $O(m)$ time by using two iterations of the Ford-Fulkerson flow-augmenting method [10]. The Ford-Fulkerson method actually finds edge-disjoint paths, but we can also compute vertex-disjoint paths after applying vertex-splitting. Specifically, we create a modified graph $\overline{G} = (\overline{V}, \overline{E})$ that results from G as follows. The vertex set $\overline{V}$ contains a pair of vertices v^- and v^+ for each vertex $v \in V$. The edge set $\overline{E}$ contains the edges (v^-, v^+) corresponding to all $v \in V$. Also, for each edge $(v, w) \in E$, we include the edge (v^+, w^-) in $\overline{E}$. It is easy to see that there are k vertex-disjoint paths from x to y in G if and only if there are k edge-disjoint paths from x^+ to y^- in $\overline{G}$. Note that $\overline{G}$ has $2n$ vertices and $m + n$ edges.

The algorithm by Cheriyan and Thurimella [5] (CT) uses matchings in order to improve the approximation guarantee of MINIMAL. Let $M \subseteq E$ be a set of edges such that every vertex has indegree and outdegree at least one in the subgraph having vertex set V and edge set M.

We call a minimum such set M a 1-*matching* of G. This can be computed in time $O(m\sqrt{n})$ via a reduction to maximum bipartite matching [18]. After computing M, the CT algorithm executes a slightly modified filtering phase, which applies the two vertex-disjoint paths test to all edges in $E \setminus M$. Hence, CT computes a subgraph $G' = (V, E')$ of G, where $E' = M \cup F$ and F is a minimal set of edges of G such that G' is 2-vertex-connected. Algorithm CT also runs in $O(m^2)$ time.

Algorithm 2: LH-Z-CT(G)

Input: 2-vertex-connected digraph $G = (V, E)$
Output: 3/2-approximation of a smallest 2-vertex-connected spanning subgraph $H = (V, E_H)$ of G

```
1  Compute a 1-matching M of G.
2  Choose an arbitrary vertex s of G as start vertex.
3  Let G' be the subgraph of G \ s, for arbitrary start vertex s, that contains only the
     edges in M.
4  Compute the strongly connected components C_1, ..., C_k in G'.
5  Form a contracted version Ğ of G \ s as follows. For each strongly connected
     component C_i of G', we contract all vertices in C_i into a representative vertex
     u_i ∈ C_i.
6  Compute a strongly connected spanning subgraph Ğ' of Ğ. Let Z be the original
     edges of G that correspond to the edges of Ğ selected in Ğ'.
7  Set H ← (V, E_H = M ∪ Z).
8  Compute a low-high order δ of flow graph G with start vertex s.
9  foreach vertex v ≠ s do
10     if there are two edges (u, v) and (w, v) in E_H such that u <_δ v and v <_δ w then
11         do nothing
12     end
13     else if there is no edge (u, v) ∈ E_H such that u <_δ v then
14         find an edge e = (u, v) ∈ E with u <_δ v
15         set E_H ← E_H ∪ {e}
16     end
17     else if there is no edge (w, v) ∈ E_H such that v <_δ w then
18         find an edge e = (w, v) ∈ E with v <_δ w or w = s
19         set E_H ← E_H ∪ {e}
20     end
21 end
22 Execute the analogous steps of lines 4–17 for the reverse flow graph G^R with start
     vertex s.
23 foreach edge (x, y) of E_H \ M do
24     if there are two vertex-disjoint paths from x to y in H \ (x, y) then
25         Set E_H ← E_H \ (x, y).
26     end
27 end
28 return H = (V, E_H)
```

Our Algorithm LH-Z-CT (whose pseudocode is described below) works as follows. First, it computes a 1-matching M as CT. Let s be an arbitrary start vertex, and let G' be the subgraph of $G \setminus s$ that contains only the edges in M. We compute the strongly connected components $C_1, \ldots, C_k$ in G', and form a contracted version $\breve{G}$ of $G \setminus s$ as follows. For each strongly connected component C_i of G', we contract all vertices in C_i into a representative vertex $u_i \in C_i$.

Then, we execute the linear-time algorithm of Zhao et al. [27] to compute a strongly connected spanning subgraph of $\breve{G}$, and store the original edges of G that correspond to the selected edges by the Zhao et al. algorithm. Let Z be this set of edges. Next, we compute

a low-high order of G with root s, and use it in order to compute a 2-vertex-connected spanning subgraph H of G using as many edges from Z and M as possible, as in LH-Z.

Then, we run the filtering phase of Cheriyan and Thurimella, as follows. For each edge (x, y) of H that is not in M, we test if x has two vertex-disjoint paths to y in $H \setminus (x, y)$. If it does, then we set $H \leftarrow H \setminus (x, y)$.

▶ **Theorem 4.** *Algorithm LH-Z-CT computes a 3/2-approximation for 2VCSS in $O(m\sqrt{n}+n^2)$ time.*

Proof. First, we note that the spanning subgraph computed by algorithm LH-Z-CT is 2-vertex-connected since it satisfies Property 1. Indeed, let H' be the graph computed in lines 1–22. Then H' is 2-vertex-connected, since it contains a strongly connected spanning subgraph of $G \setminus s$, and a set of edges that satisfies a low-high order of G and G^R. Also, the filtering phase preserves the 2-vertex-connectivity of H.

Next, we establish the 3/2 approximation ratio of LH-Z-CT by showing that a specific execution of CT produces the same output subgraph.

Let S be the set of edges of H' (i.e., the edges of H just after the execution of lines 1–22). Note that the approximation ratio of CT does not depend on the order that edges are processed during the filtering phase. Hence, we can assume that CT processes the edges of $E \setminus S$ first. Notice that for each $(x, y) \in E \setminus S$, H' contains two vertex-disjoint paths from x to y. Hence, each such edge will not be included in the subgraph computed by CT. So, if we fix the order in which the edges in S are processed, the filtering phase in both CT and LH-Z-CT will remove exactly the same redundant edges.

Finally, we consider the running time of LH-Z-CT. Line 1 takes $O(m\sqrt{n})$ time [18], and lines 2–5 take $O(m)$ time [24]. In line 6, we can compute a SCSS of $\breve{G}$ in $O(m)$ time [27], and in line 8 we can compute a low-high order of G in $O(m)$ time [14]. Finally, the loops in lines 9–2 and 23–27 take $O(m)$ and $O(n^2)$ time, respectively. ◀

We mention that in our implementation, the bipartite matching is computed via max-flow, using an implementation of the Goldberg-Tarjan push-relabel algorithm [17] from [7], which is known to be very fast in practice.

5 Empirical Analysis

For the experimental evaluation we use the graph datasets shown in Table 1, taken from the Koblenz Network Collection [22], the Stanford Network Analysis Project [23] and the 9th DIMACS Implementation Challenge [8]. For each tested graph, we computed its largest 2-vertex-connected subgraph and used that as input to our algorithms. We wrote our implementations in `C++`, using `g++ v.4.6.4` with full optimization (flag `-O3`) to compile the code. We report the running times on a GNU/Linux machine, with Ubuntu (16.04LTS): a Dell Inspiron 64-bit machine with Intel® Core ™ i7-7500U processor's seventh-generation (4 MB of cache, up to 3.5GHz) and 16 GB of RAM memory. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `getrusage` function, averaged over ten different runs.

Since we do not know the size of the optimal 2VCSS in each of these graphs, we measure the quality of the produced solution $G' = (V', E')$ by calculating the relative distance from the naive theoretical lower bound, i.e., $\frac{|E'|-2|V|}{2|V|} \times 100\%$. We refer to this relative distance as the *quality measure* (qm).

Table 1 Real-world graphs used in the experiments. From each original graph, we extracted its largest 2-vertex-connected subgraph. The number of vertices n and of edges m refer to each such subgraph.

Graph	**2VCCs**		**Type**
	n	m	
amazon-302	55414	241663	co-purchase [23]
amazon-601	276049	2461072	co-purchase [23]
advogato	3140	35979	social [22]
rome99	2249	6467	road network [8]
soc-Epinions	17117	395183	trust network [23]
web-BerkStan-1	1106	8206	web [23]
web-BerkStan-2	4927	28142	web [23]
web-BerkStan-3	29145	439148	web [23]
web-Google	77480	840829	web [23]
web-NotreDame-1	1462	10195	web [23]
web-NotreDame-2	1409	9663	web [23]
web-NotreDame-3	1416	13226	web [23]
web-Stanford-1	5179	129897	web [23]
web-Stanford-2	10893	162295	web [23]
wiki-signed	14895	324776	online contact [22]
wikiTalk	49430	2461072	wiki communication [23]

5.1 Implemented Algorithms

In our experimental evaluation we compared a total of six algorithms for computing the (approximated) smallest 2-vertex-connected spanning subgraph. In addition to our two new algorithms, LH-Z (Section 3) and LH-Z-CT (Section 4), we also considered the algorithms from [11]: FAST, CT, MINIMAL and FAST-CT.

Algorithm FAST computes a 3-approximation in linear-time by using divergent spanning trees. Specifically, it computes the edges of two divergent spanning trees of G and of two divergent spanning trees of G^R so that it satisfies Property 1(a). Then it tests if these edges also satisfy Property 1(b), and if not it adds the edges of a SCSS of $G \setminus s$ by running the algorithm of Zhao et al. [27]. MINIMAL computes a 2-approximation in $O(m^2)$ time by applying the two vertex-disjoint paths test (see Section 4) and FAST-CT combines FAST with the 3/2-approximation algorithm of Cheriyan and Thurimella [5], which gives a 3/2-approximation in $O(m\sqrt{n} + n^2)$ time. In the experiments reported in [11], FAST achieved the best running times, while FAST-CT achieved the best solution quality.

Here, we also provide a new and faster implementation of CT, MINIMAL and FAST-CT, that we refer to as CT+, MINIMAL+ and FAST-CT+. The main improvement is in the implementation of the filtering phase. In the original implementations in [11], the filtering phase is performed by computing dominators in order to avoid computing the modified graph

$\overline{G}$ (which is obtained by applying vertex-splitting). Specifically, to test if $G' \setminus (x,y)$ has two vertex-disjoint paths from x to y, FAST-CT sets x as the start vertex of G' and computes the immediate dominator $d(y)$ of y. Such two paths exist if and only if $d(y) = x$. Our new implementations apply two iterations of the flow-augmentation method on the modified graph $\overline{G}$, as described in Section 4.

(We used the same implementation of the filtering phase in LH-Z-CT as well.)

We only report the comparison of the running times for FAST-CT and its improved implementation FAST-CT+ in Figure 2 and Table 2. (Both implementations produce the same solutions.) As it is evident, our improved implementation is faster by one order of magnitude. Similar speedups are obtained for CT+ and MINIMAL-CT+

5.2 Experimental Results

The running time and quality measure of LH-Z, FAST, LH-Z-CT, FAST-CT+, CT+, and MINIMAL+ are given in Tables 3 and 4, and plotted in Figures 3 and 4. Recall that the quality measure of each of the algorithms FAST-CT, CT, and MINIMAL from [11] is identical to the quality measure achieved by the corresponding improved implementations FAST-CT+, CT+, and MINIMAL+, respectively. Hence, we do not report the results of the former implementations. It is easy to observe that the algorithms belong to two distinct classes, with FAST and LH-Z being faster than the rest by approximately four to five orders of magnitude. One the other hand, on average they produce a 2VCSS with about 10–20% more edges.

Since for large scale graphs it is important to be able to compute a good solution very fast, it is interesting to compare the performance of the linear-time algorithms FAST and LH-Z. We observe that in all test cases LH-Z was able to compute a 2VCSS with 6–25% fewer edges at the price of a small overhead in the running time.

In our next experiment, we compare algorithms FAST-CT+ and LH-Z-CT which produce the best solutions overall. Observe that LH-Z-CT is always faster, mainly due to the fact that it has to process fewer edges during the filtering phase. Moreover, in some instances LH-Z-CT is significantly faster; e.g., its more than 45% times faster for wiki-signed and amazon-601. FAST-CT+ on the other hand, produced better solutions in 10 out the 14 test graphs, but the difference is marginal (at most 6.7% fewer edges).

Finally, we consider the performance of CT+ and MINIMAL+. Notice that although MINIMAL+, rather surprisingly, computes better solutions in 3 test graphs, its performance is rather unstable. Observe, for instance, that for 3 graphs (web-Stanford-2, web-BerkStan-3 and web-Google) it computes a worse solution than LH-Z. CT+ seems more robust in that sense, but with the exception of 4 graphs it computed an inferior solution compared to FAST-CT+, while being significantly slower.

Hence, our main conclusions are the following:

- Our new linear-time algorithm, LH-Z, computes a 2VCSS of reasonable quality very fast. Hence, if one wants a fast and good solution LH-Z is the right choice.
- Executing the filtering phase on a sparse subgraph of the input digraph, produced by either FAST or LH-Z, not only decreases the running time drastically, but also helps to compute a smaller 2VCSS in the end.

Acknowledgments. We are grateful to the authors of [7] for providing the implementation of their algorithm.

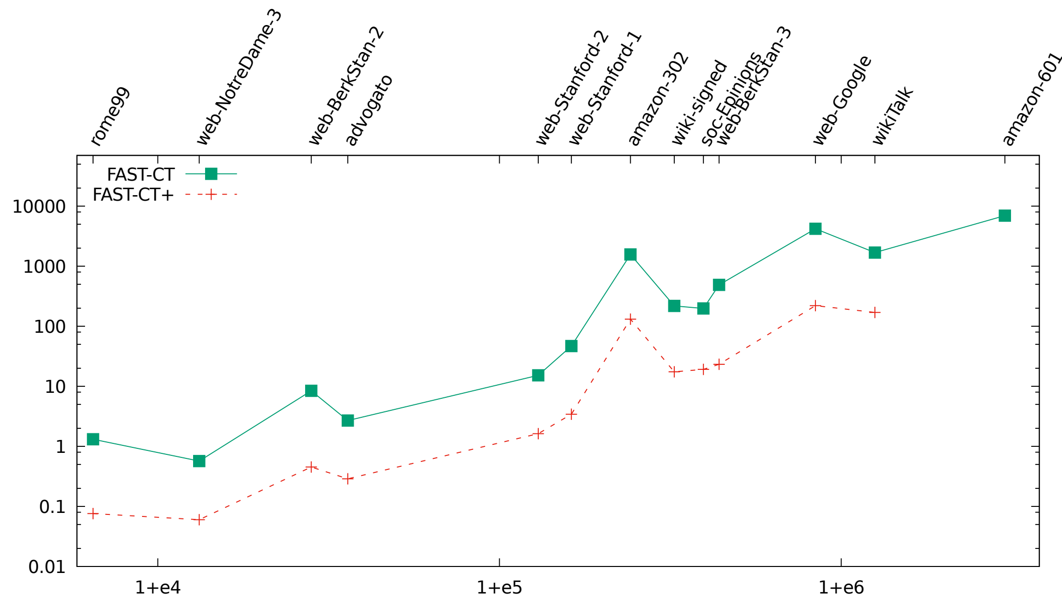


Figure 2 Running times (in seconds) of FAST-CT vs our improved implementation FAST-CT+. (The data on the experiment with amazon-601 is not reported, because FAST-CT took too long to finish.)

Table 2 Running times (in seconds) of the plot shown in Figure 2.

Graph	FAST-CT+	FAST-CT
rome99	0.076	1.312
web-BerkStan-1	0.044	0.220
web-NotreDame-3	0.060	0.568
web-BerkStan-2	0.452	8.404
advogato	0.352	2.692
web-Stanford-2	1.628	15.228
web-Stanford-1	3.424	46.800
amazon-302	131.376	1569.784
wiki-signed	17.436	217.672
soc-Epinions	19.132	197.940
web-BerkStan-3	23.264	488.604
web-Google	220.480	4200.888
wikiTalk	169.580	1689.104
amazon-601	6959.168	>24h

Table 3 Solution Quality (Quality Measure) of all algorithms that we considered.

Graph	LH-Z	FAST	LH-Z-CT	FAST-CT+	CT+	MINIMAL+
rome99	19.39	26.52	7.65	7.16	6.85	6.56
web-BerkStan-1	26.36	54.75	9.13	6.69	21.75	4.66
web-NotreDame-3	40.89	65.50	24.61	26.20	39.83	33.47
web-BerkStan-2	24.78	35.89	9.18	7.66	17.16	20.25
advogato	50.26	80.69	20.84	16.83	16.57	20.95
web-Stanford-2	40.79	73.20	14.94	14.36	32.02	47.78
web-Stanford-1	53.74	65.88	27.70	28.95	41.18	51.81
amazon-302	27.65	48.86	10.59	11.46	13.68	9.61
wiki-signed	58.37	84.44	36.61	28.02	27.50	29.80
soc-Epinions	50.50	75.63	21.98	19.86	20.59	27.56
web-BerkStan-3	51.25	70.76	35.38	32.45	41.23	52.25
web-Google	42.83	65.24	24.88	24.40	35.60	44.95
wikiTalk	62.99	78.11	40.91	38.91	37.38	38.33
amazon-601	34.82	68.99	10.16	12.74	24.64	17.08

Table 4 Running times (in seconds) of all algorithms that we considered.

Graph	LH-Z	FAST	LH-Z-CT	FAST-CT+	CT+	MINIMAL+
rome99	0.001	0.001	0.062	0.076	0.092	0.144
web-BerkStan-1	0.001	0.001	0.016	0.044	0.060	0.052
web-NotreDame-3	0.004	0.001	0.036	0.060	0.088	0.088
web-BerkStan-2	0.004	0.004	0.332	0.452	1.148	1.792
advogato	0.004	0.004	0.196	0.352	0.792	0.916
web-Stanford-2	0.012	0.008	1.380	1.628	4.640	7.668
web-Stanford-1	0.012	0.016	2.472	3.424	6.948	11.104
amazon-302	0.040	0.036	86.120	131.376	156.956	195.064
wiki-signed	0.028	0.024	9.132	17.436	42.984	66.948
soc-Epinions	0.040	0.028	13.792	19.132	58.780	79.216
web-BerkStan-3	0.028	0.024	20.072	23.264	53.524	77.900
web-Google	0.084	0.072	168.096	220.480	378.764	687.964
wikiTalk	0.116	0.092	140.464	169.580	838.528	1191.800
amazon-601	0.396	0.400	3678.768	6959.168	69434.112	11374.128

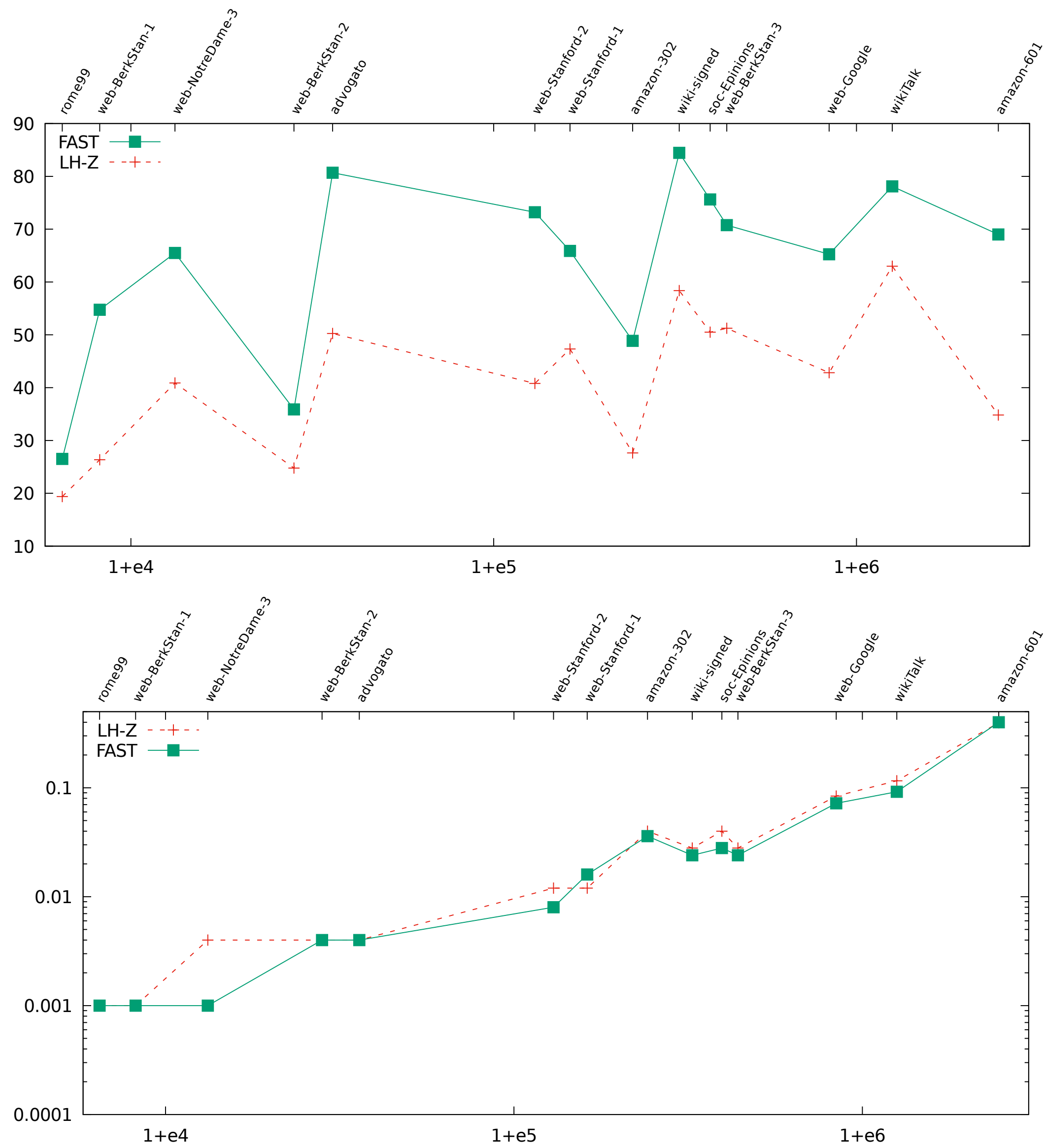


Figure 3 Performance of linear-time algorithms: solution quality (top) and running time in seconds (bottom). Running times and graph sizes (number of edges) are shown in log scale.

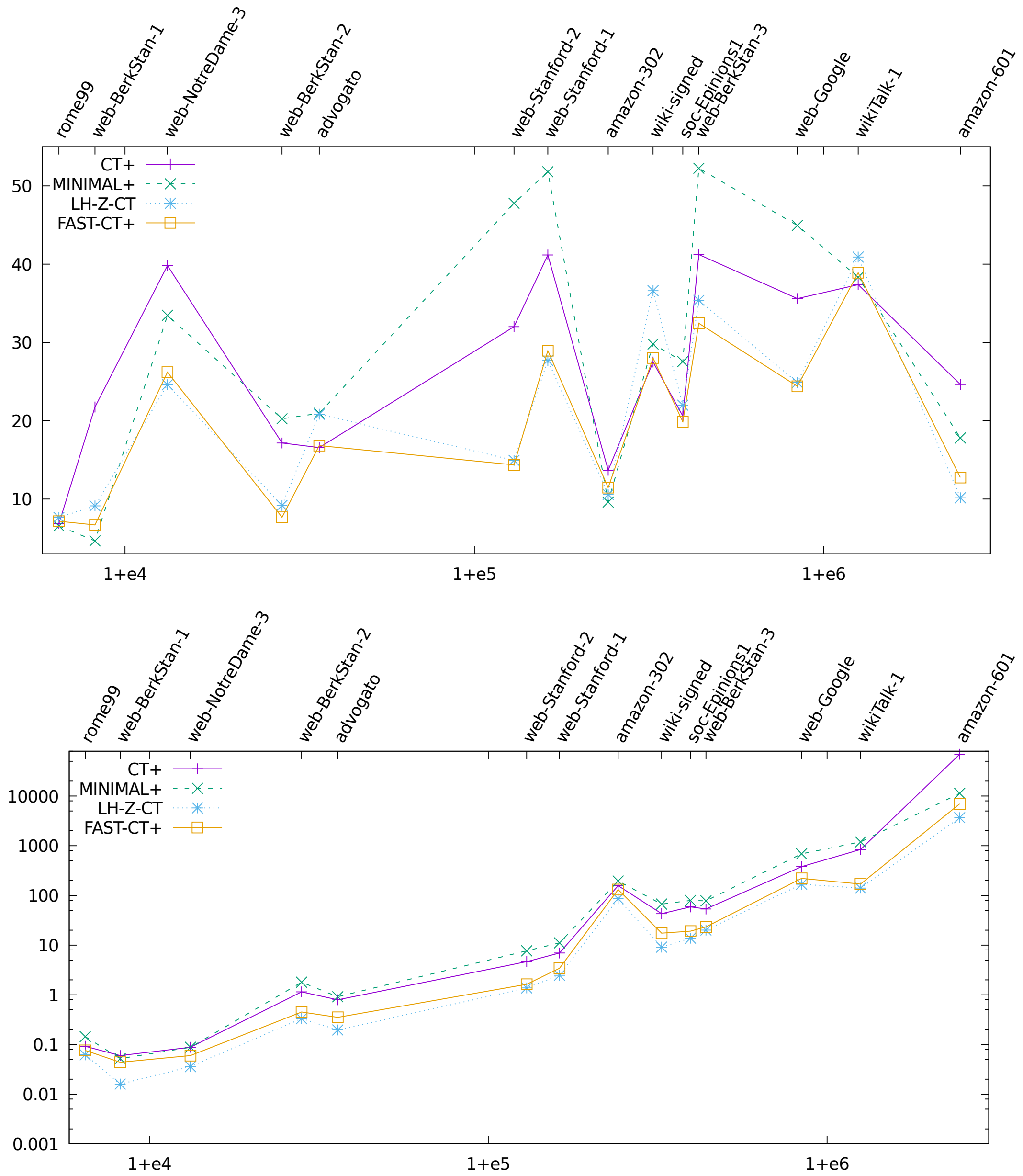


Figure 4 Performance of algorithms CT+, MINIMAL+, LH-Z-CT and FAST-CT+: solution quality (top) and running time (bottom). (Better viewed in color.) Running times and graph sizes (number of edges) are shown in log scale.

References

1 S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.
2 S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability for directed graphs. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 528–543. Springer Berlin Heidelberg, 2015.
3 S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability subgraph: Generic and optimal. In *Proc. Symposium on Theory of Computing*, pages 509–518, 2016.
4 A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.
5 J. Cheriyan and R. Thurimella. Approximating minimum-size k-connected spanning subgraphs via matching. *SIAM J. Comput.*, 30(2):528–560, 2000.
6 R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
7 D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph partitioning with natural cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, 2011.
8 C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths, 2007. URL: `http://www.dis.uniroma1.it/~challenge9/`.
9 J. Fakcharoenphol and B. Laekhanukit. An $O(\log^2 k)$-approximation algorithm for the k-vertex connected spanning subgraph problem. In *Proc. Symposium on Theory of Computing*, STOC'08, pages 153–158, New York, NY, USA, 2008. ACM.
10 D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 2010.
11 L. Georgiadis. Approximating the smallest 2-vertex connected spanning subgraph of a directed graph. In *Proc. European Symposium on Algorithms*, pages 13–24, 2011.
12 L. Georgiadis, G. F. Italiano, A. Karanasiou, C. Papadopoulos, and N. Parotsidis. Sparse subgraphs for 2-connectivity in directed graphs. In *Proc. Symposium on Experimental Algorithms*, (SEA 2016), pages 150–166, 2016.
13 L. Georgiadis, G. F. Italiano, C. Papadopoulos, and N. Parotsidis. Approximating the smallest spanning subgraph for 2-edge-connectivity in directed graphs. In *Proc. European Symposium on Algorithms*, pages 582–594, 2015.
14 L. Georgiadis and R. E. Tarjan. Dominator tree certification and divergent spanning trees. *ACM Transactions on Algorithms*, 12(1):11:1–11:42, November 2015.
15 L. Georgiadis and R. E. Tarjan. Addendum to "Dominator tree certification and divergent spanning trees". *ACM Transactions on Algorithms*, 12(4):56:1–56:3, August 2016.
16 L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications (JGAA)*, 10(1):69–94, 2006.
17 A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35:921–940, October 1988.
18 J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.
19 G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012.
20 S. Khuller, B. Raghavachari, and N. E. Young. Approximating the minimum equivalent digraph. *SIAM J. Comput.*, 24(4):859–872, 1995. Announced at *SODA 1994*, 177-186.
21 G. Kortsarz and Z. Nutov. Approximating minimum cost connectivity problems. *Approximation Algorithms and Metaheuristics*, 2007.

22 Jérôme Kunegis. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW'13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350, 2013.

23 J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

24 R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

25 T. Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theoretical Computer Science*, 465:35–48, 2012.

26 J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Proc. 2nd International Conference on Certified Programs and Proofs*, pages 27–42. Springer, 2012.

27 L. Zhao, H. Nagamochi, and T. Ibaraki. A linear time 5/3-approximation for the minimum strongly-connected spanning subgraph problem. *Information Processing Letters*, 86(2):63–70, 2003.

Extending Search Phases in the Micali-Vazirani Algorithm*

Michael Huang[1] and Clifford Stein[2]

1 **Department of IEOR, Columbia University, New York, NY, USA**
mh3166@columbia.edu
2 **Department of IEOR, Columbia University, New York, NY, USA**
cliff@ieor.columbia.edu

Abstract

The Micali-Vazirani algorithm is an augmenting path algorithm that offers the best theoretical runtime of $O(n^{0.5}m)$ for solving the maximum cardinality matching problem for non-bipartite graphs. This paper builds upon the algorithm by focusing on the bottleneck caused by its search phase structure and proposes a new implementation that improves efficiency by extending the search phases in order to find more augmenting paths. Experiments on different types of randomly generated and real world graphs demonstrate this new implementation's effectiveness and limitations.

1998 ACM Subject Classification G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases matching, graph algorithm, experimental evaluation, non-bipartite

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.10

1 Introduction

Matching, a commonly studied problem in the field of algorithms, has a variety of applications in other fields such as bioinformatics, computer science, statistics, and operations research. The paper focuses on a new efficient implementation for finding the maximum cardinality matching for non-bipartite graphs. Additionally, our experiments on non-bipartite graphs generated from historical NYC taxi cab trip data, demonstrate non-bipartite matching in solving consumer matching problems like forming carpools.

Non-bipartite matching has been a well studied problem. Since first $O(n^4)$ algorithm provided by Edmonds' [3] in 1965 there have been contributions from Gabow [5], Kameda and Munro [9], Lawler [11], Even and Kariv [4], and Micali and Vazirani [15]. The Micali-Vazirani (MV) algorithm remains the most efficient algorithm for non-bipartite graphs with the same $O(\sqrt{n}m)$ Hopcroft and Karp runtime [8] for bipartite graphs. Implementation studies from the first DIMACS Implementation Challenge [2, 14] showed that the algorithm is efficient in practice as well by comparing it against Gabow's $O(n^3)$ implementation. In a more recent study by Kececioglu and Pecqueur [10], its performance was still better against Tarjan's $O(mn\,\alpha(m,n))$ implementation, but performed slightly worse compared to the modified $O(mn\,\alpha(m,n))$ implementation presented.

In this study, we improve the MV algorithm by testing various implementations on different graph types. Since the MV algorithm is an augmenting path algorithm, previous

* This work was partially supported by NSF grant CCF-1421161.

© Michael Huang and Clifford Stein;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 10; pp. 10:1–10:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

works have suggested improving the performance by reducing the number of phases. We also reduce the number of phases. Our key idea is that, in each phase, we find extra augmenting paths in addition to the maximal set of shortest edge-disjoint augmenting paths. This additional work only takes $O(m)$ time and, in many cases, significantly reduces the number of phases of the algorithm. We also consider several previously suggested improvements and find that one, greedy initialization, also helps reduce the number of phases. Using both of these improvements we obtain significant reduction in the number of phases. While the worst case running time remains, the observed running time is much better.

2 Algorithm

2.1 Basic Definitions

A *matching* for an undirected graph $G = (V, E)$ is a set of edges M such that no two edges meet at a vertex. We designated edges and the corresponding vertices in M and not in M as matched and unmatched edges/vertices, respectively. The maximum cardinality matching problem finds any matching such that the number of edges in the set is maximized. Many algorithms that solve this problem search for augmenting paths. An *alternating path* is a simple path that alternates between edges in M and in $E - M$. An *augmenting path* is an alternating path that starts and ends with unmatched vertices. We *augment* by changing unmatched edges to matched edges and vice versa for all edges on the augmenting path–increasing the cardinality of the matching increases by one. When there are no more augmenting paths, a maximum matching is found.

Matching in bipartite and non-bipartite graphs qualitatively differ. In the first type, all lengths of alternating paths to a matched vertex v have the same *parity* (odd or even), where as matched vertices in second type can be both–which lead to odd length alternating cycles. This makes matching more difficult for non-bipartite graph, but Edmonds addressed this challenge by introducing structures called *blossoms* to handle odd-length cycles. In the MV algorithm, *petals* are the equivalent structure and are essential for achieving the efficient runtime.

2.2 Micali-Vazirani Algorithm

We will give a brief description of the Micali-Vazirani Algorithm based on Vazirani's most recent paper on the topic [16]. This paper presented a complete proof of the runtime and defined some new terms that we will use to describe the finer details of the algorithm.

2.2.1 Key Concepts

First we provide concepts specific to the MV algorithm. Below are relevant definitions that are used in the high level description of the MV algorithm in Listing 1.

▶ **Definition 1** (Evenlevel and oddlevel of v)**.** An evenlevel/oddlevel is the length of the shortest even/odd alternating path from an unmatched vertex to v, denoted as evenlevel(v) and oddlevel(v), respectively.

▶ **Definition 2** (Minlevel and maxlevel of v)**.** The minlevel of v, denoted as minlevel(v), is the minimum of evenlevel(v) and oddlevel(v). Similarly, the maxlevel is the maximum of evenlevel(v) and oddlevel(v).

Listing 1 Micali-Vazirani Algorithm pseudocode.

```
1 Set initial greedy matching for G
2 Reset edge labels
3 Set minlevel = 0 and maxlevel = ∞ for each unmatched vertex
4 Set minlevel = ∞ and maxlevel = ∞ for each matched vertex
5 Set level = 0
6 If there exist u such that maxlevel(u) == level or minlevel(u) ==
    ↪ level then continue, else go to line 31
7 For each u such that maxlevel(u) == level or minlevel(u) == level:
8   For each unscanned (u,v) with appropriate edge parity:
9     If minlevel(v) ≥ level + 1 then,
10      Set minlevel(v) = level + 1
11      Add u to the list of predecessors of u
12      Label (u,v) as prop
13    Else,
14      label (u,v) as bridge
15      If tenacity((u,v)) != ∞ then
16        Add (u,v) to the list of bridges with the same tenacity
17 For each bridge of tenacity == 2*level + 1:
18   Find support using DDFS
19   If bottleneck found then
20     Augment alternating path
21     Delete the vertices in the augmented path and all vertices that
         ↪ are orphanned (no predcessors) as a result
22   Else,
23     For each v in the support:
24       Set maxlevel(v) = 2*level + 1 - minlevel(v)
25       If v is an inner vertex then
26         For all incident (v,u) which are not props:
27           If tenacity((u,v)) != ∞ then
28             Add (u,v) to the list of bridges with the same tenacity
29 Set level = level + 1
30 If augmentation occured then go back to line 2, else go to line 6
31 Return the current matching
```

▶ **Definition 3** (Inner and outer vertices). A vertex is outer if $\text{oddlevel}(v) > \text{evenlevel}(v)$ and inner otherwise.

▶ **Definition 4** (Tenacity). Tenacity of vertex v is defined as, $\text{tenacity}(v) = \text{evenlevel}(v) + \text{oddlevel}(v)$. Tenacity of edge (u, v) is defined as, $\text{tenacity}((u, v)) = \text{oddlevel}(u) + \text{oddlevel}(v) + 1$ for an matched edge and $\text{tenacity}((u, v)) = \text{evenlevel}(u) + \text{evenlevel}(v) + 1$ for an unmatched edge.

▶ **Definition 5** (Predecessor, prop, and bridge). For any edge (u, v) such that $\text{minlevel(v)} == \text{minlevel(u)} + 1$, u is defined to be a predecessor of v. Any edge that joins a vertex and its predecessor is defined as a prop. If an edge is not a prop, then it is a bridge.

▶ **Definition 6** (Support of a bridge). If (u, v) is a bridge of tenacity t, then the support is defined as $\{w|\text{tenacity}(w) = t \text{ and } \exists \text{ a maxlevel}(w) \text{ path containing } (u, v)\}$

The *double depth first search* (ddfs) algorithm is also specifically emphasized in Vazirani's paper, since it is an essential method for finding augmenting paths and forming petals. The

ddfs finds disjoint paths to the root nodes from any pair of vertices in a level graph. In the MV algorithm, if such paths exist then an augmenting path is found, otherwise there is a *bottleneck* and we form a petal.

2.2.2 Algorithm Description

The MV algorithm is a non-bipartite matching algorithm that operates in phases. Each phase finds a maximal set of vertex disjoint shortest length augmenting paths. Like the Hopcroft-Karp algorithm for bipartite graphs, each phase synchronously constructs a level graph using breadth-first search from unmatched vertices to find alternating paths. Every time the level graph expands, the algorithm identifies bridges and performs double depth first searches on them to check for augmenting paths and to form petals. After performing the double depth first searches at the current level, the phase ends if a path was augmented. The algorithm terminates once we search the entire graph and do not find an augmenting path.

Below is a general overview of the MV algorithm in Listing 1:

- Lines 5–30: One complete phase
- Lines 6–16: The synchronous breadth-first search
- Lines 17–28: Using ddfs to find augmenting paths and forming petals
- Lines 30: Phase termination condition
- Lines 6: Algorithm termination condition

Since each phase ends when the maximum set of vertex disjoint minimum length alternating path is augmented, there are at most $\sqrt{n}$ phases. The algorithm also guarantees $O(m)$ [6] runtime per phase which leads to the overall runtime of $O(m\sqrt{n})$.

2.3 Past Work

In Kececioglu and Pecqueur [10] a new $O(mn\,\alpha(m,n))$ time implementation of Edmonds' algorithm demonstrated that including simple heuristics in the algorithm could significantly impact runtime. While the MV algorithm is theoretically the most efficient algorithm known for non-bipartite matching, experimental studies have identified potential inefficiencies in implementation [14]. Due to the heuristics' success in improving the Edmonds' algorithm, we considered three analogous heuristics and improvements to target these inefficiencies of the MV algorithm.

2.3.1 Greedy matching initialization

It is a natural idea to "initialize" a maximum matching algorithm with a greedy (maximal) matching. Kececioglu [10] considered initializing the modified Edmonds' algorithm with different greedy matching algorithms since starting with a larger matching would reduce the number of augmenting paths needed to be found. The function is the same in the MV algorithm, but its impact may be greater since we would start with fewer search trees which would reduce the amount of searching per phase in addition to reducing the total number of phases.

2.3.2 Order of bridge processing

Bridge processing potentially impacts the performance of the MV algorithm as well. Mattingly [14] provided examples demonstrating the importance of bridge processing order towards scenarios where a optimal matching could be found in a single phase. We can extend this observation and note that even if a maximum matching cannot be found in the phase,

the order processing bridges can affect phase performance. By processing bridges that lead to augmentations first, we can avoid processing future bridges that may be topologically deleted. Changing the order of bridge processing leads to not only fewer phases as Mattingly suggested, but also shorter phases.

2.3.3 Blossom formation

In previous work, Crocker [2] noted that blossom formation limited the performance of Gabow's implementation of the Edmonds' blossom algorithm. Kececioglu [10] addressed the issue with heuristics to avoid or delay blossom related operations for Tarjan's implementation of Edmonds' blossom algorithm. While these heuristics failed to provide significant performance boosts, it may improve the performance of the MV algorithm. This is closely related the order of bridge processing since we may be able to avoid forming petals if the related bridges are deleted after an augmentation.

2.4 Preliminary Results

In initial trials, only greedy matching initialization significantly reduced the number of phases of the MV algorithm. For determining the best order for processing trees, difficulty arose from identifying which alternating paths should be augmented. Attempts were made by considering the number different free nodes and alternating paths that were associated with a bridge, however, tracking that information was complicated and priority was hard to determine since we could only compare bridges associated with alternating paths of the same length. Reducing blossom formation was unsuccessful as well since phases without augmentations need blossoms to be formed in order to construct the level graph properly, meaning the gains from reducing the number of blossoms formed is diminished in future phases.

2.5 Algorithmic contribution

The Micali-Vazirani algorithm can be improved either by reducing the number of phases or improving the efficiency of the phases themselves. Due to the bottlenecks analyzed and the previous work of others, this paper mainly focuses on the former rather than the latter and achieves this by proposing broader algorithmic changes rather than technical/computational changes. As a result, the improvements in the number of phases are unaffected by the computing environment and can be directly compared to the theoretical runtime.

2.5.1 Motivation

The primary inefficiency examined in this paper relates to the termination conditions of each phase. In previous works, it has been noted that the MV algorithm's performance is closely related to the number of phases due to the high overhead in initiating phases[2]. As the minimum alternating paths get longer, the time to grow each individual search tree from the unmatched vertices increases and the number of paths augmented per phase decreases. The initialization for each phase becomes inefficient if we continually reconstruct the same trees and do not find an augmenting path.

An avenue to attack this inefficiency is to preserve the search trees by continuing the search phase. Figure 1 provides motivation for why this works. The dashed edges denote matches while the solid edges are unmatched edges. As the graph continues to the right, the number of edges separating the triangles grows so that the edges between the kth and $k+1$th triangle is greater than the number of edges between the $k-1$th and kth triangle.

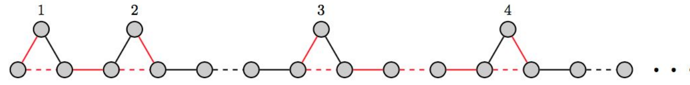


Figure 1 The red edges form the augmenting path between the free vertices at the top of each triangle.

In this scenario, the greedy initialization of the graph leaves vertices 1, 2, 3, and 4 unmatched. A phase of the base MV algorithm terminates after we augment the path between vertices 1 and 2 in the figure. Even though the search trees of vertices 3 and 4 remain unaffected by the augmentation, the phase resets and the progress towards finding the path between these two vertices is discarded. Extending the search phase would allow the augmenting path between 3 and 4 to be discovered in the next search level instead of the next phase. Assuming the numbering continues in the figure, nodes $2n-1$ and $2n$ will be matched in the same phase as well. Thus, instead of augmenting the path between the $2n-1$ and $2n$ in the nth phase, we augment the paths for all pairs in the first phase.

2.5.2 Termination conditions

Our primary contribution to the MV algorithm targets the termination bottleneck by not resetting phases even after an augmentation occurs. To describe the changes to the algorithm we reference the pseudocode in Listing 1 and the actual Python code. Below are the names of the key procedures in the Python code and the corresponding lines in the pseudocode.

- MIN – This is the search procedure that constructs the level graph and corresponds to lines 7–16 in Listing 1. The MIN procedure in the Python code is provided an array of vertices to search every phase. These vertices are set in the previous MIN procedures and in the previous MAX procedures. If the list is empty at the start of the procedure, the MV algorithm terminates.
- MAX – This is the bridge processing procedure that forms petals and augments alternating paths. It corresponds to lines 17–28 in Listing 1. When petals are formed, vertices are given their maxlabel and are added to the list of nodes for MIN to process. If augmentation occurs the current phase terminates.

In the MV algorithm, each phase alternates between running the MIN and MAX procedure until the termination condition in MIN or MAX is realized. Our modification removes the augmentation termination condition (line 30) in MAX and replaces it with the condition that the phase will terminate after running a designated number of MAX procedures that augmented at least one path. We can set this value so that each phase terminates only when the end condition is met by MIN. The extended phase MV algorithm pseudocode can be seen in Listing 2 in the appendix.

The modification allows the algorithm to preserve work put into constructing the level graph in the current phase. Additionally, the level graph that remains after augmentation preserves the search trees from unmatched nodes that were disjoint from the augmented paths since it topologically deletes the search trees of the matched nodes. This results in the selection of a maximal set of augmenting paths during each phase.

The three principles/observations that make the modification immediately attractive are:

1. It is guaranteed to not have more phases than the original algorithm.
2. In earlier phases, the alternating paths are shorter. That means the total number of alternating paths that intersect with augmented paths is lower and the maximal matching will find a larger fraction of the remaining matches.

3. In later phases, the number of alternating paths becomes more sparse as the number of free vertices decreases and are more likely to have different lengths. It also becomes less likely that augmenting one path will impact other search trees. Extending the phases saves in reconstructing unused search trees in a manner similar to the motivational example discussed earlier.

3 Experiments

In past experimental studies, the best algorithm in practice was Kececioglu's modification of Tarjan's $O(mn\,\alpha(m,n))$ algorithm, which ran roughly 4 times faster than Crocker's MV algorithm[10]. Our experiments will primarily focus on measuring the relative improvements that can be made on the MV algorithm by reducing the number of phases. The relative measure can provide insight on how it will perform against Kececioglu's algorithm.

Since the main goal focuses on methods reducing the number of search phases, less effort was put into the implementation and selection of data structures that could optimize the performance of each individual search phase. In order to remain consistent, we use the same implementation of the search phase for each variant of the algorithm.

The implementation was written in Python 2.7.10 and utilizes the time, csv, NumPy, and NetworkX modules. The majority of the experiments were conducted on Columbia Business School research grid. For larger graphs that required more memory to store the node and edge data, we used a Intel Xeon E5-2667 processor with 256 GB of main memory. The different implementations and experiments are available at:

`https://github.com/mh3166/Extended_MV_algorithm`

3.1 Variants/Implementations

For our experiment, we present two different modifications:

1. The first chooses a different initialization method. We construct the initial maximal matching using the heuristics and reductions discussed in Magun's experimental work with greedy matching algorithms [13]. Like our simple greedy algorithm, every time an edge is added to the the matching, the adjacent vertices and the adjacent edges are removed from the graph. We then find the maximal matching for the remaining graph. The new greedy algorithm performs 1 reductions which means that in the current graph, the edges of vertices of degree 1 must be included in the maximal matching. It also uses a heuristic (referred to as Heuristic 1), which states that each edge we add to the maximal matching must include a node with minimum degree in the current graph, and another heuristic, which states that the opposite node to the one in Heuristic 1 must have minimum degree among the neighboring vertices.
2. The second is our algorithmic contribution that replaces the termination condition for the MAX phase of the MV algorithm with a more relaxed version that only terminates a phase after it encounters a specified number of MAX phases that augmented a path. In the experiment, this number was 100 and was never reached in any trial.
   Let the following be notation for the different variants of the MV algorithm.
   a. **MV0** is the base algorithm
   b. **MV1** is the variant with modification **1.**
   c. **MV2** is the variant with modification **2.**
   d. **MV3** is the variant with modification **1.** and modification **2.**

3.2 Graphs

Building upon the previous MV algorithm work of Crocker [2] and Mattingly and Ritchey [14], we examined the graphs of previous experiments that had non-trivial processing times. Given that we know the structure of these graphs, the results can provide insight into the improved performance. We also tested the algorithm on real world graphs that provided significant challenge to the original algorithm in order to demonstrate the improved algorithm's robustness. Below are the selected graph:

1. **Random graphs** with n nodes and with expected degree d. This is a Erdős-Rényi graph, a binomial graph that was generated with the fast_gnp_ random_graph method in NetworkX [7]. It takes the inputs n and $p = \frac{d}{n}$. Following Crocker [2], we chose the average degrees that differed by factors of $2^{1/16}$ and chose a range of $0 \leq d \leq 8$.
2. **Grid graphs** with n^2 nodes with expected degree d. The grid graph is a $n \times n$ lattice graph that has some of the edges removed. Rather than having each node be adjacent to four edges, we construct it so that the average degree d of the graph is $2 \leq d \leq 4$. The graph is constructed by visiting each node and adding each of the four edges to the graph with probability $p = 1 - (1 - \frac{d}{4})^{1/2}$ which accounts for visiting each edge twice. We test sizes for $4 \leq n \leq 10$.
3. **One-connected triangles** with 2^k triangles. The graph has 2^k vertex disjoint triangles. These triangles are then interconnected by only one edge. To construct the graph, we add the following edges

$$(3i, 3i+1), (3i+1, 3i+2), (3i+2, 3i) \quad 0 \leq i \leq 2^k - 1,$$

$$(3i + (i \bmod 3), 3i + 3 + (i \bmod 3)) \quad 0 \leq i \leq 2^k - 2,$$

   where $\{0, \ldots, 3 \cdot 2^k - 1\}$ is the set of vertices. Since the algorithm will process the edges and nodes in numerical order, the graphs are randomized by switching numbering while maintaining the same structure. We test for $10 \leq k \leq 20$.
4. **Three-connected triangles** with 2^k triangles. The graph also has 2^k vertex disjoint triangles. The triangles are now interconnected by three edges instead of one. To construct the graph, we add the following edges

$$(3i, 3i+1), (3i+1, 3i+2), (3i+2, 3i) \quad 0 \leq i \leq 2^k - 1,$$

$$(3i, 3i+3), (3i+1, 3i+4), (3i+2, 3i+5) \quad 0 \leq i \leq 2^k - 2,$$

   where $\{0, \ldots, 3 \cdot 2^k - 1\}$ is the set of vertices. Again we randomize the graph by randomizing the labels of the nodes. We test for $10 \leq k \leq 14$. The lower range is because we run into recursion issues when opening petals in graphs with $k \geq 15$.
5. **Real World Graphs** are composed of selections from Stanford's Large Network Dataset Collection [12] as well as graphs constructed by the NYC taxi cab data from 2015 as shown in Table 1. From Stanford's data set we chose network graphs showing representing Amazon's product co-purchasing network as of certain dates. Most of these graphs were only selected for their size and average degree rather than for their practical application. From the NYC taxi cab data, we constructed a more realistic matching problem the graph by finding taxicab passengers who were close in departure time and location and were headed in a similar direction. See the github link for the code that generated these graphs. The maximum matching in this graph would provide the maximum number of carpooling opportunities in NYC in a day.

Table 1 Description of Real World Graphs.

Network Graph	$\|N\|$	$\|E\|$	Avg. Deg.
Amazon Co-Purchasing 3/2/03	262111	899792	3.43
Amazon Co-Purchasing 3/12/03	400727	2349869	5.86
Amazon Co-Purchasing 5/5/2003	410236	2439437	5.95
Amazon Co-Purchasing 6/1/2003	403394	2443408	6.06
2/1/15 Taxi Data	325109	952974	2.93
2/2/15 Taxi Data	569599	1487866	2.61
2/4/15 Taxi Data	1216990	3414986	2.81
2/5/15 Taxi Data	1578057	4564025	2.89
2/7/15 Taxi Data	2335680	6993447	2.99

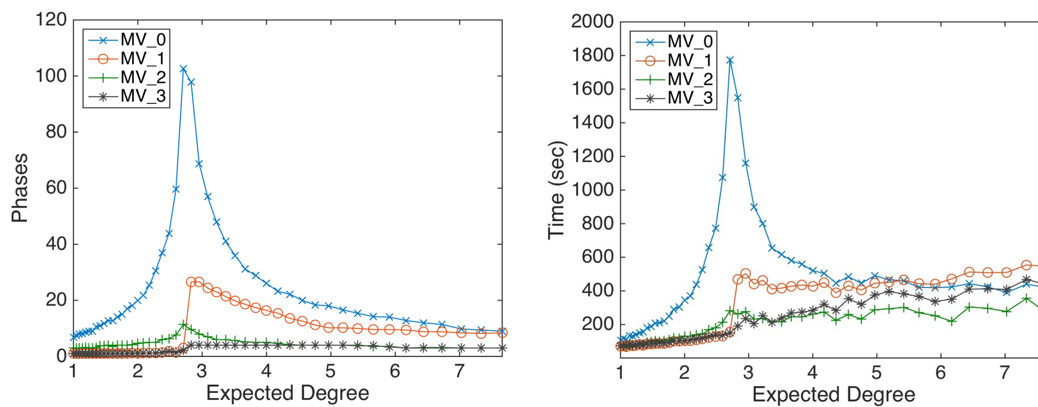


Figure 2 Average search phases and running time as the expected degree changes for random graphs with 2^{20} nodes.

4 Results

In this section, we observe the relationship between the structure of the graphs and the performance of the algorithms.

Overall, we see phase reduction and thus lower running time when utilizing our modification of extending phases. Changing the greedy initialization also has significant impact on phases (~60% reduction at best) and running time as well, but we get the most consistent performance in phase reduction when combining the two modifications as seen by MV3. In the few cases where MV3 performs worse than MV2, we are mainly hampered by overhead in our modified greedy initialization.

While we see overall improvement, we also see noticeable differences in performance on different graphs.

4.1 Random Graphs

The random graphs experiment provides a starting point to analyze what could reduce the number of phases in the MV algorithm. As seen in Figure 2, we replicated the observation from Crocker [2] that the number of phases peaks at degree ~3 and empirically observed the proposition by Bast[1] that with high probability the length of the maximum augmenting path–and thus the maximum number of phases–for any non-maximum matching is $O(\log n)$

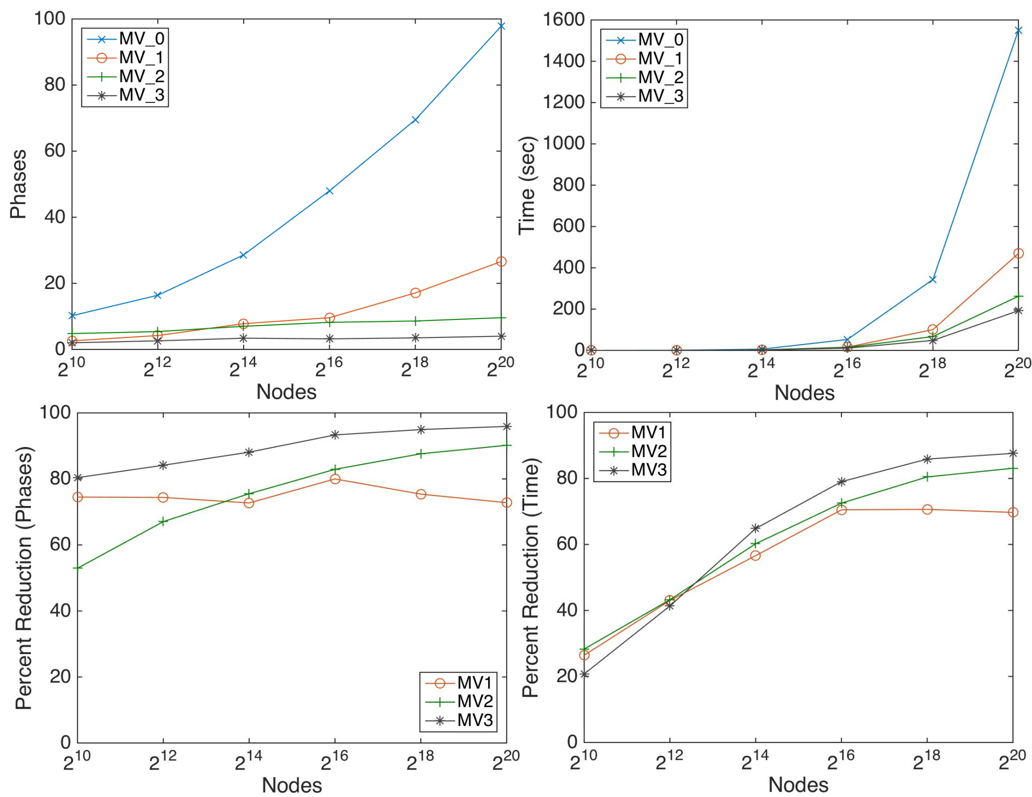


Figure 3 Average percent reduction of search phases and running time as the number of nodes increases for random graphs with an expected degree of $\sqrt{8}$.

for random graphs with average degree above some constant c. In this experiment, we also discovered that utilizing an initialization heuristic alone can reduce phases and that it can also improve the performance of the MV algorithm with extended search phases.

From Magun [13], we know that compared to initializing with a random maximal matching, both the starting number of matches and average degree of the free vertices are more likely to be greater. That implies the MV algorithm could be receiving a performance boost from the fewer matches to be found or the larger number of alternating paths per free vertex.

4.2 Grid graphs

To further investigate the effect of search phase extension and the initialization heuristic, we can compare the random graph experiment to the grid graph experiment by comparing Figure 4 to Figure 3. In the worst case examples, grid graphs require significantly more phases and thus more time to be solved compared to random graphs. Additionally, the initialization heuristic has little effect in improving performance as graph size grows. While we again see that the worst case performance occurs in randomly generated graphs with an average degree of 3, it is clear that the limited range of degrees for vertices has a negative effect on the performance of the MV algorithm.

4.3 One-connected Triangles

One-connected triangle graphs provide an even more extreme scenario where simple structures result in high phase costs for the MV algorithm. In Figure 5, we see that applying the

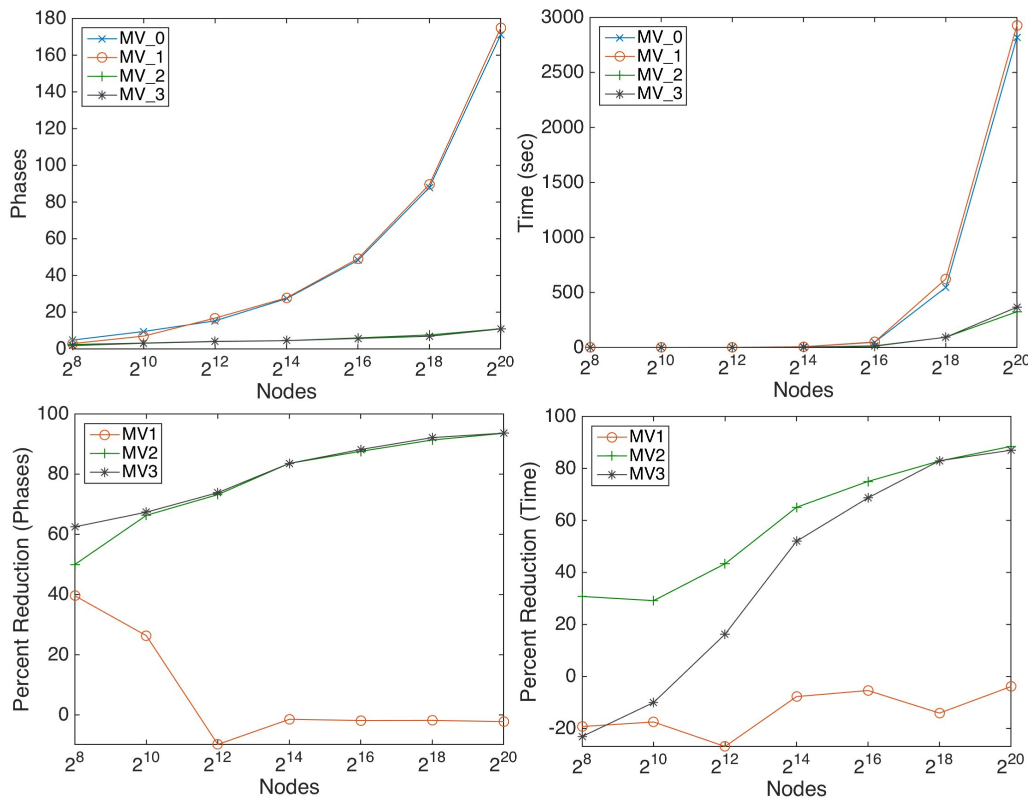


Figure 4 Average percent reduction of search phases and running time as the number of nodes increases for grid graphs with expected degree of 3.12.

initialization heuristic helps deal with that issue by consistently reducing the number of phases by 60%. However, by extending phases, the MV2 and MV3 algorithm perform significantly better by reducing the number of phases by over a factor of 50 as seen in Table 2. This type of graph closely resembles the motivational graph in Figure 1 and demonstrates the type of graphs and subgraphs that benefit from extending search phases. See the appendix for further discussion.

4.4 Three-connected Triangles

The Three-connected triangle graph experiment had similar results to the one-connected version in terms of the phase count. We include this experiment to demonstrate the effect of nested petals on performance. In the algorithm, the nested structure requires recursive calls to open petals in order to find augmenting paths. From a technical standpoint, in larger graphs this triggers the default maximum recursion depth safeguard for Python which terminates the algorithm early. This problem can be avoided by implementing a non-recursive method for finding augmenting paths as discussed by Mattingly [14].

4.5 Real World Graphs

In Table 3, we see that the benefits of the greedy initialization and phase extension apply to real world graphs. Our results primarily focused on solving the maximum cardinality matching problem for large graphs where each phase is computationally expensive. The

Table 2 Average runtime and phase for one connected triangle graphs as number of triangles increase.

	Average Runtime (sec)				Average Phases			
Triangles	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
2^{10}	0.923	0.374	0.34	0.327	24.1	10	4	3.7
2^{12}	7.734	2.627	2.019	2.04	50.5	18	4.8	4.2
2^{14}	89.298	24.088	11.175	11.002	103.1	35.6	5.4	5.1
2^{16}	829.863	195.934	53.978	52.98	200.8	70.2	6.3	5.8
2^{18}	6185.386	1668.073	238.776	221.504	410.2	140.1	8.4	6.5

Table 3 Search phases and running time for different real world graphs.

	Average Runtime (sec)				Average Phases			
Network Graph	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
Amazon Co-Purchasing 3/2/03	178.89	194.36	66.23	118.12	21	20	4	5
Amazon Co-Purchasing 3/12/03	398.23	466.23	227.72	248.53	18	19	5	3
Amazon Co-Purchasing 5/5/2003	380.5	422.09	191.73	256.7	18	17	5	4
Amazon Co-Purchasing 6/1/2003	400.7	469.1	260.45	287.07	17	18	5	4
2/1/15 Taxi Data	622.61	458.32	187.15	141.46	53	42	10	6
2/2/15 Taxi Data	972.02	946.72	315.67	257.57	56	48	11	7
2/4/15 Taxi Data	3142.8	2852.5	724.79	640.93	72	59	10	7
2/5/15 Taxi Data	5277.9	4552.8	1158.1	1035.1	75	63	10	7
2/7/15 Taxi Data	9480.2	8593.6	1868.8	1546.5	81	75	11	7

results of the real world graphs fall in line with the artificially created graphs. The Amazon Co-Purchasing graphs demonstrate the higher average degree correlates to fewer phases observation discovered in the random graph experiments. The modifications to the MV algorithm thus have a lesser impact towards improving performance. The taxi graph results provide an example of graphs whose performance is dictated by degree and structure. While the higher degree reduces the number of phases needed compared to that of the worst case random graphs, the number of phases needed does not decrease at the same rate. Thus, we see significant improvement similar to that of artificial graphs that have a specific structure. These real world graphs show that our modifications are effective against structure that hampers the MV algorithm.

5 Discussion

5.1 Runtime

Through our experiments we have seen that the removal of the termination condition in MAX greatly reduces the number of phases the algorithm must be processed. When we plot the growth of number of phases vs. the log size of the problem in Figure 6, we see that there is a linear relationship. Thus, it seems that the number of phases grows in $O(\log n)$.

Additionally, in our experiments, after obtaining the maximum matching, we also were able to calculate the fraction of remaining matches that were processed during each phase. Figure 7 plots the average fraction found per phase for one-triangle connected graphs as the graph size grows in number of triangles. We see that the algorithms with the extended phases can better preserve the $O(\log n)$ number of phases by consistently finding a high fraction of remaining matches per phase.

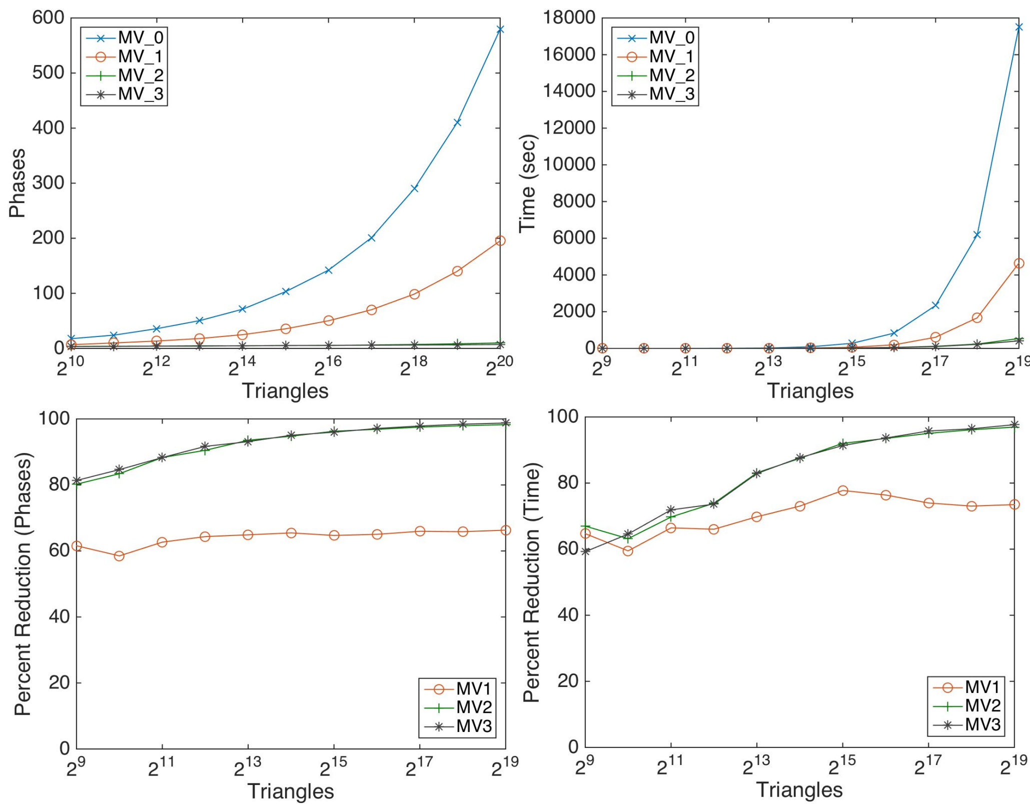


Figure 5 Average percent reduction of search phases and running time as the number of triangles increases for one-connected triangle graphs.

5.2 Worst Case Graph

From our experiments we have only seen graphs that perform well after making the algorithmic modification to the MV algorithm. Figure 8 is an example of a type of graph where the worst case runtime is $O(\sqrt{n}m)$.

With the greedy matching shown after the initial phase, both the base and the modified algorithm will have a $O(\sqrt{n}m)$ runtime if it first augments the path that intersects all other alternating paths. The resulting new graph again has a alternating path that intersects all other alternating paths. In each phase, there is the possibility that only one path will be augmented. Since we start with n matchings to be found and the starting graph is of size $O(n^2)$, we obtain the $O(\sqrt{n})$ number of phases. In this case, the modified algorithms have the same performance as the base MV algorithm.

6 Conclusion

We introduced a new implementation of the Micali-Vazirani algorithm that effectively reduced the number of search phases and demonstrated its effectiveness on a variety of graph types. Our primary contribution considers extending search phases which in our experiment reduces the number of phases by at least 40% in smaller graphs and up to 98% in larger graphs. The phase reduction translates well to runtime when the phase improvement is large. This saves in overhead cost from reseting the graph each phase. For cases where the graphs could already be solved by the base MV algorithm in a few phases, runtime improvement is

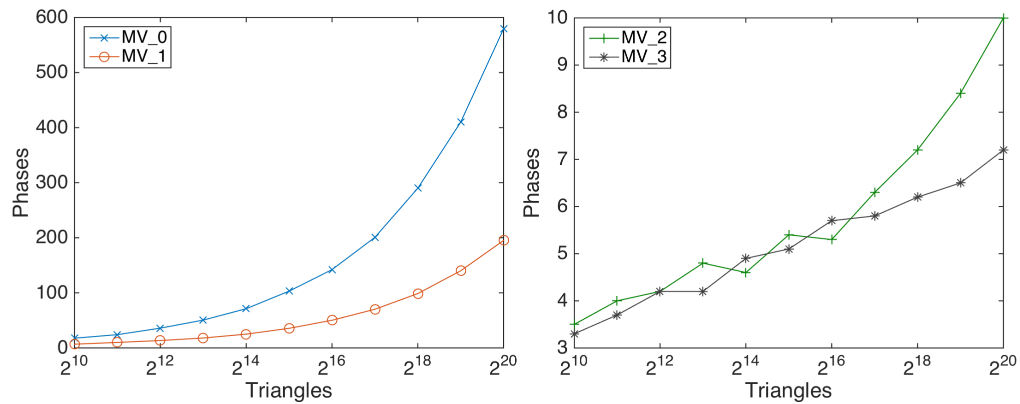


Figure 6 Comparison of phase growth with and without the termination condition in MAX phase.

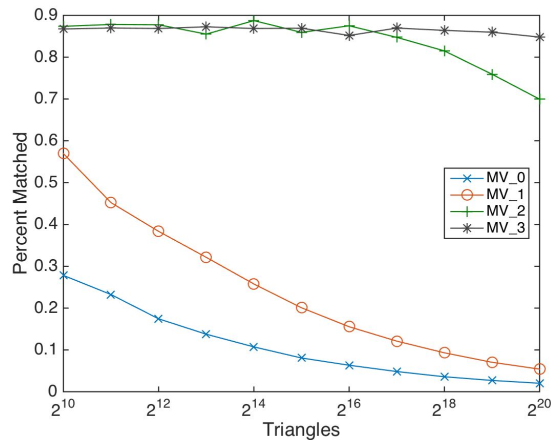


Figure 7 Comparison of average percent of remaining matches found per phase.

smaller since the algorithm is already using each phase efficiently. In the future, we hope to improve the more technical aspects of the implementation, such as choosing more efficient data structures and cleaning up any inefficiencies in the code. A fully optimized version would allow for better comparison with matching algorithms in previous works as well as matching algorithms exclusive to bipartite graphs. From a more theoretical standpoint, we would also like to study how the structure of the graphs impacts the performance of the modified MV algorithm. This may provide insight that could result in more significant contributions that could improve the performance of the MV algorithm.

References

1 Holger Bast, Kurt Mehlhorn, Guido Schafer, and Hisao Tamaki. Matching algorithms are fast in sparse random graphs. *Theory of Computing Systems*, 39(1):3–14, 2006.

2 Steven T. Crocker. An experimental comparison of two maximum cardinality matching programs. In Catherine C. McGeoch David S. Johnson, editor, *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *Discrete Mathematics and Theoretical Computer Science*, pages 519–537, 1993.

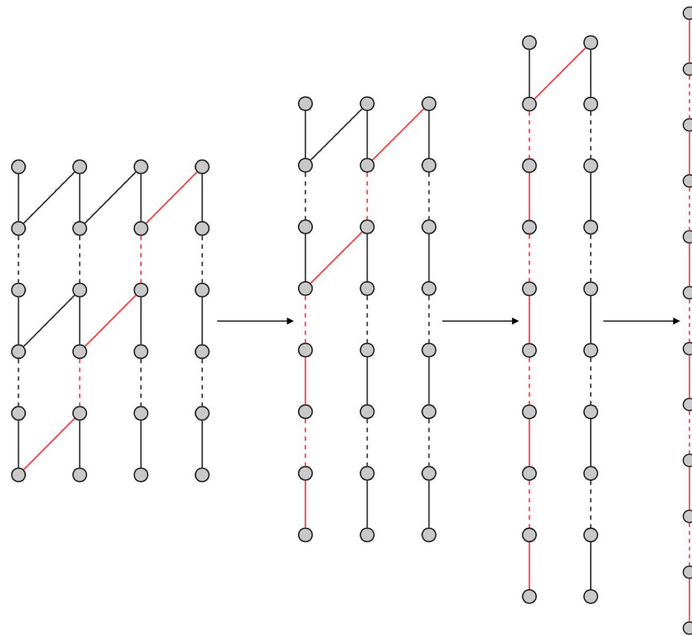


Figure 8 Example of $O(\sqrt{n}m)$ performance for the modified algorithm: Augmented paths are in red. In the example, the worst case maximal set of one is chosen in each phase.

3 Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

4 Shimon Even and Oded Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 100–112. IEEE, 1975.

5 Harold N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM (JACM)*, 23(2):221–234, 1976.

6 Harold N. Gabow. Set-merging for the Matching Algorithm of Micali and Vazirani. *arXiv preprint arXiv:1501.00212*, 2014.

7 Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

8 John E. Hopcroft and Richard M. Karp. A $n^{5/2}$ algorithm for maximum matchings in bipartite. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 122–125. IEEE, 1971.

9 T. Kameda and I. Munro. A $O(|V| \cdot |E|)$ algorithm for maximum matching of graphs. *Computing*, 12(1):91–98, 1974.

10 John D. Kececioglu and A. Justin Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Algorithm Engineering*, pages 121–132, 1998.

11 Eugene L. Lawler. Combinatorial optimization, 1976.

12 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, jun 2014. URL: `http://snap.stanford.edu/data`.

13 Jakob Magun. Greeding matching algorithms, an experimental study. *J. Exp. Algorithmics*, 3, sep 1998. `doi:10.1145/297096.297131`.

14 R. Bruce Mattingly and Nathan P. Ritchey. Implementing an $o(sqrtNm)$ cardinality matching algorithm. In Catherine C. McGeoch David S. Johnson, editor, *Network Flows and*

Matching: First DIMACS Implementation Challenge, volume 12 of *Discrete Mathematics and Theoretical Computer Science*, pages 539–556, 1993.

15 Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algoithm for finding maximum matching in general graphs. In *Foundations of Computer Science, 1980, 21st Annual Symposium on*, pages 17–27. IEEE, 1980.

16 Vijay V. Vazirani. An improved definition of blossoms and a simpler proof of the MV matching algorithm. *CoRR*, abs/1210.4594, 2012. URL: `http://arxiv.org/abs/1210.4594`.

A Appendix

A.1 Additional Results: Graphs

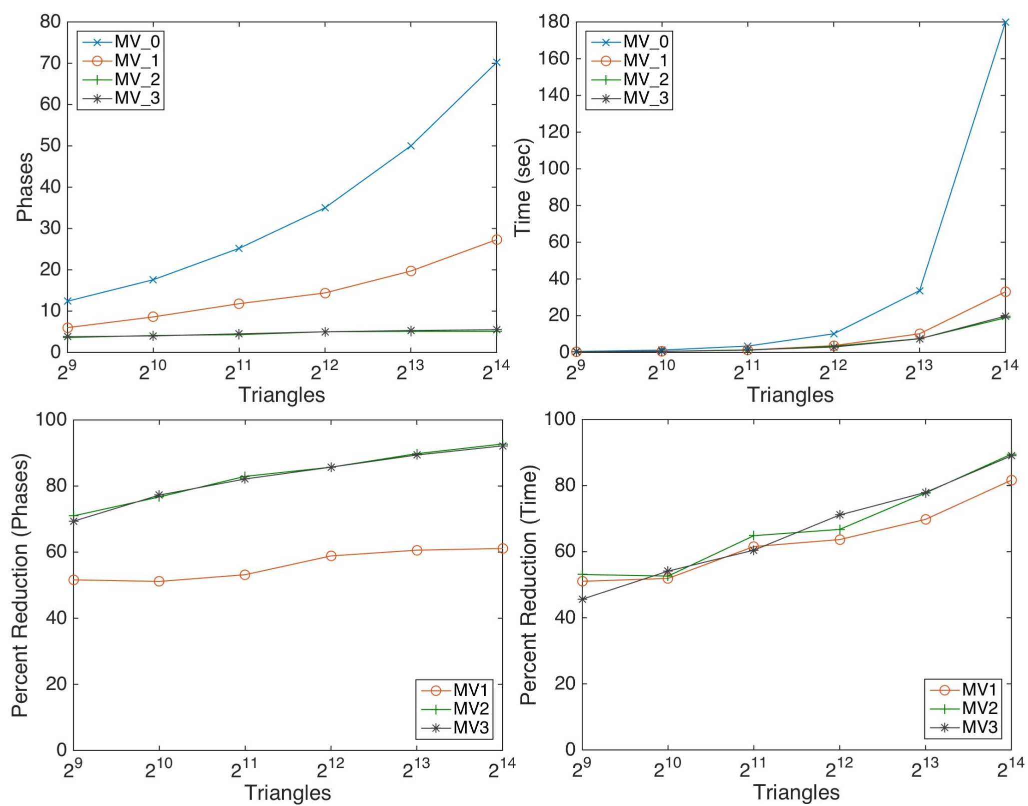


Figure 9 Average percent reduction of search phases and running time as the number of triangles increases for three-connected triangle graphs.

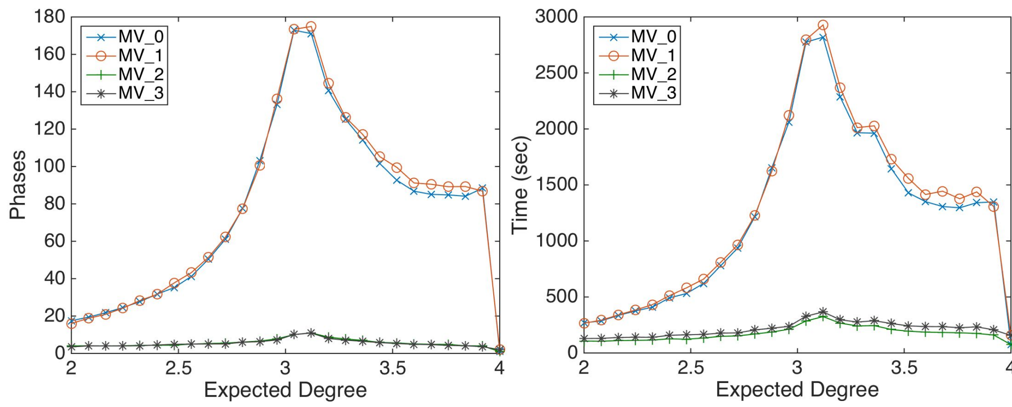


Figure 10 Average search phases and running time as the expected degree increases for grid graphs with 2^{20} nodes.

A.2 Additional Results: Tables

Table 4 Average search phases and running time as the number of nodes increases for random graphs with an expected degree of $\sqrt{8}$.

	Average Runtime (sec)				Average Phases			
N	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
2^{10}	0.106	0.078	0.076	0.084	10.2	2.6	4.8	2
2^{12}	0.846	0.482	0.48	0.496	16.4	4.2	5.4	2.6
2^{14}	6.208	2.694	2.47	2.184	28.6	7.8	7	3.4
2^{16}	53.052	15.654	14.58	11.13	48	9.6	8.2	3.2
2^{18}	343.739	100.947	67.186	48.578	69.5	17.1	8.6	3.5
2^{20}	1549.718	469.49	262.22	192.17	97.8	26.6	9.6	4

Table 5 Average runtime and phase for three connected triangle graphs as number of triangles increase.

	Average Runtime (sec)				Average Phases			
Triangles	MV0	MV1	MV2	MV3	MV0	MV1	MV2	MV3
2^{9}	0.535	0.262	0.251	0.291	12.4	6	3.6	3.8
2^{10}	1.266	0.609	0.6	0.581	17.6	8.6	4.1	4
2^{11}	3.464	1.332	1.218	1.37	25.5	11.8	4.3	4.5
2^{12}	10.144	3.686	3.374	2.931	35	14.4	5	5
2^{13}	33.639	10.169	7.482	7.423	50	19.7	5.1	5.3
2^{14}	179.793	33.029	18.856	19.772	70.2	27.3	5.1	5.5

Table 6 Average search phases and running time as the number of nodes increases for grid graphs with expected degree of 3.12.

N	Average Runtime (sec) MV0	MV1	MV2	MV3	Average Phases MV0	MV1	MV2	MV3
2^{10}	0.12	0.141	0.085	0.132	4.8	2.9	2.4	1.8
2^{12}	0.639	0.811	0.362	0.535	9.5	7	3.2	3.1
2^{14}	6.082	6.552	2.125	2.919	15.3	16.8	4.1	4
2^{16}	47.305	49.861	11.824	14.806	27.4	27.8	4.5	4.5
2^{18}	545.662	622.501	93.476	92.991	87.9	89.5	7.6	6.9
2^{20}	2819.185	2928.456	324.903	366.417	171.1	174.9	11	10.9

A.3 Micali-Vazirani Algorithm with Extended Phases

Listing 2 Micali-Vazirani Algorithm pseudocode.

```
1  Set initial greedy matching for G
2  Reset edge labels
3  Set minlevel = 0 and maxlevel = ∞ for each unmatched vertex
4  Set minlevel = ∞ and maxlevel = ∞ for each matched vertex
5  Set level = 0
6  Set augmentation_count = 0
7  If there exist u such that maxlevel(u) == level or minlevel(u) ==
       ↪ level then continue, else go to line 31
8  For each u such that maxlevel(u) == level or minlevel(u) == level:
9    For each unscanned (u,v) with appropriate edge parity:
10     If minlevel(v) ≥ level + 1 then,
11       Set minlevel(v) = level + 1
12       Add u to the list of predecessors of u
13       Label (u,v) as prop
14     Else,
15       label (u,v) as bridge
16 If tenacity((u,v)) != ∞ then
17   Add (u,v) to the list of bridges with the same tenacity
18   For each bridge of tenacity == 2*level + 1:
19     Find support using DDFS
20     If bottleneck found then
21       Augment alternating path
22       Delete the vertices in the augmented path and all vertices that
             ↪ are orphanned (no predcessors) as a result
23     Else,
24       For each v in the support:
25         Set maxlevel(v) = 2*level + 1 - minlevel(v)
26         If v is an inner vertex then
27           For all incident (v,u) which are not props:
28             If tenacity((u,v)) != ∞ then
29               Add (u,v) to the list of bridges with the same tenacity
30 Set level = level + 1
31 If augmentation occured then augmentation_count += 1
32 If augmentation_count == 100 then go to line 2, else go to line 7
33 Return the current matching
```

A.4 Lemmas

▶ **Lemma 7.** *Augmenting a maximal set of vertex disjoint augmenting paths that includes a maximal set of vertex disjoint shortest augmenting paths increases the length of the shortest augmenting path in the new matching*

Proof. Let M be the initial matching, let M' be the subsequent matching after augmenting the maximal set of disjoint shortest augmenting paths P in M, and let M'' be the subsequent matching after augmenting the maximal set of disjoint augmenting paths P' in M. From Hopcroft and Karp [8] we know that if the length of the shortest augmenting path of a matching M is l, then the length of the shortest augmenting path in M' is strictly greater than l. Since $P \in P'$ and the paths in $P' - P$ are disjoint from P, augmenting the set $P' - P$ in matching M' gives us M''. The shortest augmenting path cannot get shorter, thus the shortest augmenting path in M'' is still strictly greater than l. ◀

A.5 Additional Discussion

We can show $O(m \log n)$ in the worst case for certain families of graphs while also proving the base algorithm operates in $O(\sqrt{n}m)$ time. The construction of such a graph is similar to that of the one-connected triangles graph in that it is constructed by joining vertex disjoint triangles. Instead of joining the triangles with one edge, it is replaced by a sequence of edges described below.

Let the number of edges between each triangle be determined by the following. Let us number the triangles from 1 to k. The number of edges joining triangle $2i-1$ and $2i$ be $2(i-1)+1$ and let the number of edges joining triangle $2i$ and $2i+1$ be $2i+1$.

To demonstrate the worst case performance, let us assume that after a greedy matching, we have a graph such that the only free vertices are the nodes of each triangle i that is not adjacent to the set of edges connecting the triangle to triangle $i-1$ or $i+1$.

In the case of the modified algorithm MV2, we see that after the initial greedy matching, each phase selects a maximal set of augmenting paths. Since the unmatched vertices essentially lie on the same line, augmenting paths will always be guaranteed to be matched at most two potential free vertices. That means the algorithm also never encounters non vertex disjoint alternating paths. The problem can be reduced to selecting a maximal matching, where unmatched edges are the alternating paths. Rather than randomly choosing the matching, it is determined by the length of the alternating paths, but since it is a maximal selection, the equivalent problem is also a maximal matching. Since we know that a maximal matching is a 2-approximation of the maximum matching, it implies that the maximal set of alternating paths is also a 2-approximation of selecting the maximum set of augmenting paths. Thus, we are guaranteed to reduce the number of remaining matchings to be found in half each phase giving us the $O(m \log n)$ worse case run time.

A Framework of Dynamic Data Structures for String Processing*

Nicola Prezza

Technical University of Denmark, DTU Compute, Lyngby, Denmark
npre@dtu.dk

Abstract

In this paper we present DYNAMIC, an open-source C++ library implementing dynamic compressed data structures for string manipulation. Our framework includes useful tools such as searchable partial sums, succinct/gap-encoded bitvectors, and entropy/run-length compressed strings and FM indexes. We prove close-to-optimal theoretical bounds for the resources used by our structures, and show that our theoretical predictions are empirically tightly verified in practice. To conclude, we turn our attention to applications. We compare the performance of five recently-published compression algorithms implemented using DYNAMIC with those of state-of-the-art tools performing the same task. Our experiments show that algorithms making use of dynamic compressed data structures can be up to three orders of magnitude more space-efficient (albeit slower) than classical ones performing the same tasks.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases C++, dynamic, compression, data structure, bitvector, string

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.11

1 Introduction

Dynamism is an extremely useful feature in the field of data structures for string manipulation, and has been the subject of study in many recent works [5, 17, 24, 31, 20, 13]. These results showed that – in theory – it is possible to match information-theoretic upper and lower bounds on space of many problems related to dynamic data structures while still supporting queries in provably optimal time. From the practical point of view however, many of these results are based on complicated structures which prevent them from being competitive in practice. This is due to several factors that in practice play an important role but in theory are often poorly modeled, such as cache locality, branch prediction, disk accesses, context switches, memory fragmentation. Good implementations must take into account all these factors in order to be practical; this is the main reason why little work in this field has been done on the experimental side. An interesting and promising (but still under development) step in this direction is represented by Memoria [22], a C++14 framework providing general purpose dynamic data structures. Other libraries are also still under development (ds-vector [7]) or have been published but the code is not available [5, 17, 2]. Practical works considering weaker dynamic queries have also appeared. In [29] the authors consider rewritable arrays of integers (no indels or partial sums are supported). In [30] practical close-to-succinct dynamic tries are described (in this case, only navigational and child-append operations are supported). To the best of our knowledge, the only working implementation of a dynamic succinct bitvector

* Part of this work was done while the author was a PhD student at the University of Udine, Italy. Work supported by the Danish Research Council (DFF-4005-00267).

© Nicola Prezza;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 11; pp. 11:1–11:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is [11]. This situation changes dramatically if the requirement of dynamism is dropped. In recent years, several excellent libraries implementing static data structures have been proposed: `sdsl` [12] (probably the most comprehensive, used, and tested), `pizza&chili` [25] (compressed indexes), `sux` [34], `succinct` [33], `libcds` [18]. These libraries proved that *static* succinct data structures can be very practical in addition to being theoretically appealing.

In view of this gap between theoretical and practical advances in the field, in this paper we present `DYNAMIC`: a C++11 library providing practical implementations of some basic succinct and compressed dynamic data structures for string manipulation: searchable partial sums, succinct/gap-encoded bitvectors, and entropy/run-length compressed strings and FM indexes. Our library has been extensively profiled and tested, and offers structures whose performance are provably close to the theoretical lower bounds (in particular, they approach succinctness and logarithmic queries). `DYNAMIC` is an open-source project and is available at [8].

We conclude by discussing the performance of five recently-published BWT/LZ77 compression algorithms [26, 28, 27] implemented with our library. On highly compressible datasets, our algorithms turn out to be up to three orders of magnitude more space-efficient than classical algorithms performing the same tasks.

2 The `DYNAMIC` library

The core of our library is a searchable partial sum with inserts data structure (SPSI in what follows). We start by formally defining the SPSI problem and showing how we solve it in `DYNAMIC`. We then proceed by describing how we use the SPSI structure as a building block to obtain the dynamic structures implemented in our library.

2.1 The Core: Searchable Partial Sums with Inserts

The Searchable Partial Sums With Inserts (SPSI) problem asks for a data structure PS maintaining a sequence $s_1, \ldots, s_m$ of non-negative integers and supporting the following operations on it:

- $\texttt{PS.sum}(i) = \sum_{j=1}^{i} s_j$;
- $\texttt{PS.search}(x)$ is the smallest i such that $\sum_{j=1}^{i} s_j > x$;
- $\texttt{PS.update}(i, \delta)$: update s_i to $s_i + \delta$. δ can be negative as long as $s_i + \delta \geq 0$;
- $\texttt{PS.insert}(i)$: insert 0 between s_{i-1} and s_i (if $i = 0$, insert in first position).

As discussed later, a consequence of the fact that our SPSI does not support `delete` operations is that also the structures we derive from it do not support `delete`; we plan to add this feature in our library in the future.

`DYNAMIC`'s SPSI is a B-tree storing integers $s_1, \ldots, s_m$ in its leaves and subtree size/partial sum counters in internal nodes. SPSI's operations are implemented by traversing the tree from the root to a target leaf and accessing internal nodes' counters to obtain the information needed for tree traversal. The choice of employing B-trees is motivated by the fact that a big node fanout translates to smaller tree height (w.r.t. a binary tree) and nodes that can fully fit in a cache line (i.e. higher cache efficiency). We use a leaf size l (i.e. number of integers stored in each leaf) always bounded by

$$0.5 \log m \leq l \leq \log m$$

and a node fanout $f \in \mathcal{O}(1)$. f should be chosen according to the cache line size; a bigger value for f reduces cache misses and tree height but increases the asymptotic cost of handling

single nodes. See Section 2.2 for a discussion on the maximum leaf size and f values used in practice in our implementation. Letting $l = c \cdot \log m$ be the size of a particular leaf, we call the coefficient $0.5 \le c \le 1$ the *leaf load*.

In order to improve space usage even further while still guaranteeing very fast operations, integers in the leaves are packed contiguously in word arrays and, inside each leaf $\mathcal{L}$, we assign to each integer the bit-size of the largest integer stored in $\mathcal{L}$. In the next section we prove that this simple blocking strategy leads to a space usage very close to the information-theoretic minimum number of bits needed to store the integers $s_1, \dots, s_m$. It is worth to notice that – as opposed to other works such as [2] – inside each block we use a fixed-length integer encoding. Such an encoding allows much faster queries than variable-length integer codes (such as, e.g., Elias' delta or gamma) as in our strategy integers are stored explicitly and do not need to be decoded first. Whenever an integer overflows the maximum size associated to its leaf (after an `update` operation), we re-allocate space for all integers in the leaf. This operation takes $\mathcal{O}(\log m)$ time, so it does not asymptotically increase the cost of `update` operations. Crucially, in each leaf we allocate space only for the integers actually stored inside it, and re-allocate space for the whole leaf whenever we insert a new integer or we split the leaf. With this strategy, we do not waste space for half-full leaves[1]. Note, moreover, that since the size of each leaf is bounded by $\Theta(\log m)$, re-allocating space for the whole leaf at each insertion does not asymptotically slow down `insert` operations.

2.1.1 Theoretical Guarantees

Let us denote with $m/\log m \le L \le 2m/\log m$ the total number of leaves, with $\mathcal{L}_j$, $0 \le j < L$, the j-th leaf of the B-tree (using any leaf order), and with $I \in \mathcal{L}_j$ an integer belonging to the j-th leaf. The total number of bits stored in the leaves of the tree is

$$\sum_{0 \le j < L} \sum_{I \in \mathcal{L}_j} \mathit{max_bitsize}(\mathcal{L}_j)$$

where $\mathit{max_bitsize}(\mathcal{L}_j) = \max_{I \in \mathcal{L}_j}(\mathit{bitsize}(I))$ is the bit-size of the largest $I \in \mathcal{L}_j$, and $\mathit{bitsize}(x)$ is the number of bits required to write number x in binary: $\mathit{bitsize}(0) = 1$ and $\mathit{bitsize}(x) = \lfloor \log_2 x \rfloor + 1$, for $x > 0$. The above quantity is equal to

$$\sum_{0 \le j < L} c_j \cdot \log m \cdot \mathit{max_bitsize}(\mathcal{L}_j)$$

where $0.5 \le c_j \le 1$ is the j-th leaf load. Since leaves' loads are always upper-bounded by 1, the above quantity is upper-bounded by

$$\log m \sum_{0 \le j < L} \mathit{max_bitsize}(\mathcal{L}_j)$$

which, in turn, is upper-bounded by

$$\log m \sum_{0 \le j < L} \mathit{bitsize}\left(\sum_{I \in \mathcal{L}_j} I\right) \le \log m \sum_{0 \le j < L} 1 + \log_2\left(1 + \sum_{I \in \mathcal{L}_j} I\right).$$

In the above inequality, we use the upper-bound $\mathit{bitsize}(x) \le 1 + \log_2(1 + x)$ to deal with the case $x = 0$. Let $M = m + \sum_{i=1}^{m} s_i = m + \sum_{0 \le j < L} \sum_{I \in \mathcal{L}_j} I$ be the sum of all integers

[1] in practice, to speed up operations we allow a small fraction of the leaf to be empty.

stored in the structure plus m. From the concavity of log and from $L \leq 2m/\log m$, it can be derived that the above quantity is upper-bounded by

$$2m \cdot (\log(M/m) + \log\log m + 1) \ .$$

To conclude, we store $\mathcal{O}(1)$ pointers/counters of $\mathcal{O}(\log M)$ bits each per leaf and internal node. We obtain:

▶ **Theorem 1.** *Let $s_1, \ldots, s_m$ be a sequence of m non-negative integers and $M = m + \sum_{i=1}^{m} s_i$. The partial sum data structure implemented in `DYNAMIC` takes at most*

$$2 \cdot m \left(\log(M/m) + \log\log m + \mathcal{O}(\log M/\log m)\right)$$

bits of space and supports `sum`, `search`, `update`, and `insert` operations on the sequence $s_1, \ldots, s_m$ in $\mathcal{O}(\log m)$ time.

Our implementation uses the standard C++ memory allocator to allocate memory for the growing dynamic structures. As shown in the experimental section, this choice results in a non-negligible fraction of memory being wasted due to memory fragmentation. Ad-hoc allocators such as the one discussed in [5] can significantly alleviate this effect. In our experiments we observed that – even taking into account memory fragmentation – the bit-size of our dynamic partial sum structure is well approximated by function $1.19 \cdot m\left(\log(M/m) + \log\log m + \log M/\log m\right)$. See the experimental section for full details.

2.2 Plug and Play with Dynamic Structures

The SPSI structure described in the previous section is used as a building block to obtain all dynamic structures of our library. In `DYNAMIC`, the SPSI structure's type name is `spsi` and is parametrized on three template arguments: the leaf type (here, the type `packed_vector` is always used[2]), the leaf size and the node fanout. `DYNAMIC` defines two SPSI types with two different combinations of these parameters:

```
typedef spsi<packed_vector,256,16> packed_spsi;
typedef spsi<packed_vector,8192,16> succinct_spsi;
```

The reasons for the particular values chosen for the leaf size and node fanout will be discussed later. We use these two types as basic components in the definition our structures.

2.2.1 Gap-Encoded Bitvectors

`DYNAMIC` implements gap-encoded bitvectors using an SPSI to encode gap lengths: bitvector $0^{s_1-1}10^{s_2-1}1 \ldots 0^{s_m-1}1$ ($s_i > 0$) is encoded with a partial sum on the sequence $s_1, \ldots, s_m$. For space reasons, we do not describe how to reduce the gap-encoded bitvector problem to the SPSI problem; the main idea is to reduce bitvector's `access` and `rank` operations to SPSI's `search` operations, bitvector's `select` operations to SPSI's `sum` operations, bitvector's $\texttt{insert}_1$ operations to SPSI's `insert` operations, and bitvector's $\texttt{insert}_0/\texttt{delete}_0$ operations to SPSI's `update` operations.

`DYNAMIC`'s name for the dynamic gap-encoded bitvector class is `gap_bitvector`. The class is a template on the SPSI type. We plug `packed_spsi` in `gap_bitvector` as follows:

[2] `packed_vector` is simply a packed vector of fixed-size integers supporting all SPSI operations in linear time.

```
typedef gap_bitvector<packed_spsi> gap_bv;
```

and obtain:

▶ **Theorem 2.** *Let $B \in \{0,1\}^n$ be a bit-sequence with b bits set. The dynamic gap-encoded bitvector* `gap_bv` *implemented in* `DYNAMIC` *takes at most*

$$2 \cdot b\,(\log(n/b) + \log\log b + \mathcal{O}(\log n/\log b))\,(1 + o(1))$$

bits of space and supports `rank`*,* `select`*,* `access`*,* `insert`*, and* `delete`$_0$ *operations on B in $\mathcal{O}(\log b)$ time.*

In our experiments, the optimal node fanout for the SPSI structure employed in this component turned out to be 16, while the optimal leaf size 256 (these values represented a good compromise between query times and space usage). Our benchmarks show (see the experimental section for full details) that the bit-size of our dynamic gap-encoded bitvector is well approximated by function $1.19 \cdot b\,(\log(n/b) + \log\log b + \log n/\log b)$.

It is worth to notice that an alternative efficient implementation of bitvectors is run-length encoding (RLE): a bitvector $0^{k_1}1^{k_2}0^{k_3}\dots$ can be represented with an SPSI on the integer sequence $k_1, k_2, k_3, \dots$. This representation results advantageous in cases where the underlying bitvector contains also long runs of bits set (e.g. a wavelet tree on a string with long equal-letter runs). We preferred using gap-encoding for two main reasons. First of all, in our library gap-encoded bitvectors are at the core of run-length encoded strings (more details in Section 2.2.3). In such structures, every equal-letter run is marked with a bit set in a gap-encoded bitvector. This breaks the symmetry between zeros and ones in the bitvector as strings with long equal-letter runs will generate bitvectors with very few bits set. Then, note that the above-mentioned RLE bitvector representation allows for efficient `access` operations, but does not support (fast) `rank`. To support `rank`, one should store separately the cumulated lengths of runs of zeros (or ones). This is exactly what our run-length compressed string (on the alphabet $\{0,1\}$) does (see Section 2.2.3).

2.2.2 Succinct Bitvectors and Entropy-Compressed Strings

Let n be the bitvector length. Dynamic succinct bitvectors can be implemented using an SPSI where all $m = n$ stored integers are either 0 or 1. At this point, `rank` operations on the bitvector correspond to `sum` on the partial sum structure, and `select` operations on the bitvector can be implemented with `search` on the partial sum structure[3]. `access` and `insert` operations on the bitvector correspond to exactly the same operations on the partial sum structure. Note that in this case we can accelerate operations in the leaves by a factor of $\log n$ by using constant-time built-in bitwise operations such as `popcount`, masks and shifts. This allows us to use bigger leaves containing $\Theta(\log^2 n)$ bits, which results in a total number of internal nodes bounded by $\mathcal{O}(n/\log^2 n)$. The overhead for storing internal nodes is therefore of $o(n)$ bits. Moreover, since in the leaves we allocate only the *necessary* space to store the bitvector's content (i.e. we do not allow empty space in the leaves), it easily follows that the dynamic bitvector structure implemented in `DYNAMIC` takes $n + o(n)$ bits of space and supports all operations in $\mathcal{O}(\log n)$ time.

[3] Actually, `search` permits to implement only `select`$_1$. `select`$_0$ can however be easily simulated with the same solution used for `search` by replacing each integer $x \in \{0,1\}$ with $1 - x$ at run time. This solution does not increase space usage.

In our experiments, the optimal node fanout for the SPSI stucture employed in the succinct bitvector structure turned out to be 16, while the optimal leaf size 8192. `DYNAMIC`'s name for the dynamic succinct bitvector is `succinct_bitvector`. The class is a template on the SPSI type. `DYNAMIC` defines its dynamic succinct bitvector type as:

```
typedef succinct_bitvector<succinct_spsi> suc_bv;
```

We obtain:

▶ **Theorem 3.** *Let $B \in \{0,1\}^n$ be a bit-sequence. The dynamic succinct bitvector data structure `suc_bv` implemented in `DYNAMIC` takes $n + o(n)$ bits of space and supports `rank`, `select`, `access`, and `insert` operations on B in $\mathcal{O}(\log n)$ time.*

In our experiments (see the experimental section) the size of our dynamic succinct bitvector was always upper-bounded by $1.23 \cdot n$ bits. The 23% overhead on top of the optimal size comes mostly from memory fragmentation (16%). The remaining 7% comes from succinct structures on top of the bit-sequence.

Dynamic compressed strings are implemented with a wavelet tree built upon dynamic succinct bitvectors. We explicitly store the topology of the tree ($\mathcal{O}(|\Sigma| \log n)$ bits) instead of encoding it implicitly in a single bitvector. This choice is space-inefficient for very large alphabets, but reduces the number of `rank/select` operations on the bitvector(s) with respect of a wavelet tree stored as a single bitvector. `DYNAMIC`'s compressed strings (wavelet trees) are a template on the bitvector type. `DYNAMIC` defines its dynamic string type as:

```
typedef wt_string<suc_bv> wt_str;
```

The user can choose at construction time whether to use a Huffman, fixed-size, or Gamma encoding for the alphabet. Gamma encoding is useful when the alphabet size is unknown at construction time. The Huffman encoding of the string uses at most $n(H_0 + 1)$ bits; a Huffman-shaped wavelet tree only adds a low-order overhead on top of this representation. In our library, we store the Huffman tree topology using pointers (instead of concatenating the wavelet tree's bitvectors into a single bitvector). This strategy reduces the number of operations needed to navigate the tree, but adds a $\mathcal{O}(|\Sigma| \log n)$-bits overhead. We obtain:

▶ **Theorem 4.** *Let $S \in \Sigma^n$ be a string with zero-order entropy equal to H_0. The Huffman-compressed dynamic string data structure `wt_str` implemented in `DYNAMIC` takes*

$$n(H_0 + 1)(1 + o(1)) + \mathcal{O}(|\Sigma| \log n)$$

bits of space and supports `rank`, `select`, `access`, and `insert` operations on S in average $\mathcal{O}((H_0 + 1) \log n)$ time.

When a fixed-size encoding is used (i.e. $\lceil \log_2 |\Sigma| \rceil$ bits per character), the structure takes $n \log |\Sigma|(1 + o(1)) + \mathcal{O}(|\Sigma| \log n)$ bits of space and supports all operations in $\mathcal{O}(\log |\Sigma| \cdot \log n)$ time.

2.2.3 Run-Length Encoded Strings

To run-length encode a string $S \in \Sigma^n$, we adopt the approach described in [32]. We store one character per run in a string $H \in \Sigma^r$, we mark the end of the runs with a bit set in a bit-vector $V_{all}[0, \dots, n-1]$, and for every $c \in \Sigma$ we store all c-runs lengths consecutively in a bit-vector V_c as follows: every m-length c-run is represented in V_c as $0^{m-1}1$.

▶ **Example 5.** Let $S = bc\#bbbbccccbaaaaaaaaaaa$. We have: $H = bc\#bcba$, $V_{all} = 11100010001100000000001$, $V_a = 00000000001$, $V_b = 100011$, $V_c = 10001$, and $V_\# = 1$

By encoding H with a wavelet tree and gap-compressing all bitvectors, we achieve run-length compression. It can be easily shown that this representation allows supporting `rank`, `select`, `access`, and `insert` operations on S, but for space reasons we do not give these details here. In DYNAMIC, the run-length compressed string type `rle_string` is a template on the gap-encoded bitvector type (bitvectors V_{all} and V_c, $c \in \Sigma$) and on the dynamic string type (run heads H). We plug the structures of the previous sections in the above representation as follows:

```
typedef rle_string<gap_bv, wt_str> rle_str;
```

and obtain:

▶ **Theorem 6.** *Let $S \in \Sigma^n$ be a string with r_S equal-letter runs. The dynamic run-length encoded string data structure `rle_str` implemented in DYNAMIC takes*

$$r_S \cdot (4\log(n/r_S) + \log|\Sigma| + 4\log\log r_S + \mathcal{O}(\log n/\log r_S))\,(1 + o(1)) + \mathcal{O}(|\Sigma|\log n)$$

bits of space and supports `rank`, `select`, `access`, and `insert` operations on S in $\mathcal{O}(\log|\Sigma| \cdot \log r_S)$ time.

2.2.4 Dynamic FM Indexes

An FM index [10] is a data structure supporting `rank` operations over the Burrows-Wheeler transform [3] (BWT) of the text, plus a suitable sampling mechanism that associates text positions to a subset of BWT positions (i.e. a suffix array sampling). Such a data structure takes space close to that of the compressed text (provided that the string structure used is compressed) and supports fast counting and locating occurrences of a pattern in the text. If the data structure used to represent the string supports also `insert` operations, then the FM index support also left-extensions of the text [4, 21, 20].

We obtain dynamic FM indexes by combining a dynamic Burrows-Wheeler transform with a sparse dynamic vector storing the suffix array sampling. In DYNAMIC, the BWT is a template class parametrized on the L-column and F-column types. For the F column, a run-length encoded string is always used. DYNAMIC defines two types of dynamic Burrows-Wheeler transform structures (wavelet-tree/run-length encoded):

```
typedef bwt<wt_str,rle_str> wt_bwt;
typedef bwt<rle_str,rle_str> rle_bwt;
```

Dynamic sparse vectors are implemented inside the FM index class using a dynamic bitvector marking sampled BWT positions and a dynamic sequence of integers (an SPSI) storing non-null values. We combine a Huffman-compressed BWT with a succinct bitvector and an SPSI:

```
typedef fm_index<wt_bwt, suc_bv, packed_spsi> wt_fmi;
```

and obtain:

▶ **Theorem 7.** *Let $S \in \Sigma^n$ be a string with zero-order entropy equal to H_0, $P \in \Sigma^m$ a pattern occurring occ times in T, and k the suffix array sampling rate. The dynamic Huffman-compressed FM index `wt_fmi` implemented in DYNAMIC takes*

$$n(H_0 + 2)(1 + o(1)) + \mathcal{O}(|\Sigma|\log n) + (n/k)\log n$$

bits of space and supports:

- *`access` to BWT characters in average $\mathcal{O}((H_0+1)\log n)$ time*
- *`count` in average $\mathcal{O}(m(H_0+1)\log n)$ time*
- *`locate` in average $\mathcal{O}((m+occ\cdot k)(H_0+1)\log n)$ time*
- *text `left-extension` in average $\mathcal{O}((H_0+1)\log n)$ time*

If a plain alphabet encoding is used, all (H_0+1) terms are replaced by $\log|\Sigma|$ and times become worst-case.

If, instead, we combine a run-length compressed BWT with a gap-encoded bitvector and an SPSI as follows:

```
typedef fm_index<rle_bwt, gap_bv, packed_spsi> rle_fmi;
```

we obtain:

▶ **Theorem 8.** *Let $S\in\Sigma^n$ be a string whose BWT has r runs, $P\in\Sigma^m$ a pattern occurring occ times in T, and k the suffix array sampling rate. The dynamic run-length compressed FM index `rle_fmi` implemented in `DYNAMIC` takes*

$$r\cdot(4\log(n/r)+\log|\Sigma|+4\log\log r+\mathcal{O}(\log n/\log r))\,(1+o(1))+\mathcal{O}(|\Sigma|\log n)+(n/k)\log n$$

bits of space and supports:

- *`access` to BWT characters in $\mathcal{O}(\log|\Sigma|\cdot\log r)$ time*
- *`count` in $\mathcal{O}(m\cdot\log|\Sigma|\cdot\log r)$ time*
- *`locate` in $\mathcal{O}((m+occ\cdot k)(\log|\Sigma|\cdot\log r))$ time*
- *text `left-extension` in $\mathcal{O}(\log|\Sigma|\cdot\log r)$ time*

The suffix array sample rate k can be chosen at construction time.

3 Experimental Evaluation

We start by presenting detailed benchmarks of our gap-encoded and succinct bitvectors, that are at the core of all other library's structures. We then turn our attention to applications: we compare the performance of five recently-published compression algorithms implemented with `DYNAMIC` against those of state-of-the-art tools performing the same tasks and working in uncompressed space. All experiments were performed on a `intel core i7` machine with 12 GB of RAM running Linux Ubuntu 16.04.

3.1 Benchmarks: Succinct and Gap-Encoded Bitvectors

We built 34 gap-encoded (`gap_bv`) and 34 succinct (`suc_bv`) bitvectors of length $n=500\cdot10^6$ bits, varying the frequency b/n of bits set in the interval $[0.0001, 0.99]$. In each experiment, we first built the bitvector by performing n $\texttt{insert}_b$ queries, b being equal to 1 with probability b/n, at uniform random positions. After building the bitvector, we executed n $\texttt{rank}_0$, n $\texttt{rank}_1$, n $\texttt{select}_0$, n $\texttt{select}_1$, and n `access` queries at uniform random positions. Running times of each query were averaged over the n repetitions. We measured memory usage in two ways: (i) internally by counting the total number of bits allocated by our procedures – this value is denoted as *allocated* memory in our plots –, and (ii) externally using the tool `/usr/bin/time` – this value is denoted as *RSS* in our plots (Resident Set Size).

Working space. We fitted measured RSS memory with the theoretical predictions of Section 2.1.1 using a linear regression model. Parameters of the model were inferred using the statistical tool `R` (function `lm`). In detail, we fitted RSS memory in the range $b/n \in [0, 0.1]$[4] with function $k \cdot f(n, b) + c$, where: $f(n, b) = b \cdot (\log(n/b) + \log\log b + \log n/ \log b)$ is our theoretical prediction (recall that memory occupancy of our gap-encoded bitvector should never exceed $2f(n, b)$), k is a scaling factor accounting for memory fragmentation and average load distribution in the B-tree, and c is a constant accounting for the weight of loaded C++ libraries (this component cannot be excluded from the measurements of the tool `/usr/bin/time`). Function `lm` provided us with parameters $k = 1.19$ and $c = 28, 758, 196$ bits $\approx 3.4MB$. The value for c was consistent with the space measured with b/n close to 0.

Figures 1 and 2 show memory occupancy of `DYNAMIC`'s bitvectors as a function of the frequency b/n of bits set. In Figure 1 we compare both bitvectors. In Figure 2 we focus on the behavior of our gap-encoded bitvector in the interval $b/n \in [0, 0.1]$. In these plots we moreover show the growth of function $1.19 \cdot f(n, b) + 28, 758, 196$. Plot in Figure 1 shows that our theoretical prediction fits almost perfectly the memory usage of our gap-encoded bitvector for $b/n \leq 0.7$. The plot suggests moreover that for $b/n \geq 0.1$ it is preferable to use our succinct bitvector rather than the gap-encoded one. As far as the gap-encoded bitvector is concerned, memory fragmentation[5] amounts to approximately 15% of the allocated memory for $b/n \leq 0.5$. This fraction increases to 24% for b/n close to 1. We note that RSS memory usage of our succinct bitvector never exceeds $1.29n$ bits: the overhead of $0.29n$ bits is distributed among (1) `rank/select` succinct structures ($\approx 0.07n$ bits) (2) loaded C++ libraries (a constant amounting to approximately 3.4 MB, i.e. $\approx 0.06n$ bits in this case), and memory fragmentation ($\approx 0.16n$ bits). Excluding the size of C++ libraries (which is constant), our bitvector's size never exceeds $1.23n$ bits (being $1.20n$ bits on average).

Query times. Plots in Figures 3-6 show running times of our bitvectors on all except $\texttt{rank}_0$ and $\texttt{select}_0$ queries (which were very close to those of $\texttt{rank}_1$ and $\texttt{select}_1$ queries, respectively). We used a linear regression model (inferred using `R`'s function `lm`) to fit query times of our gap-encoded bitvector with function $c + k \cdot \log b$. Query times of our succinct bitvector were interpolated with a constant (with n fixed). These plots show interesting results. First of all, our succinct bitvector supports extremely fast ($0.01\mu s$ on average) `access` queries. `rank` and `select` queries are, on average, 15 times slower than `access` queries. As expected, `insert` queries are very slow, requiring – on average – 390 times the time of `access` queries and 26 times that of `rank/select` queries. On all except `access` queries, running times of our gap-encoded bitvector are faster than (or comparable to) those of our succinct bitvector for $b/n \leq 0.1$. Combined with the results depicted in Plot 1, these considerations confirm that for $b/n \leq 0.1$ our gap-encoded bitvector should be preferred to the succinct one. `access`, `rank`, and `select` queries are all supported in comparable times on our gap-encoded bitvector ($\approx 0.05 \cdot \log b$ μs), and are one order of magnitude faster than `insert` queries.

[4] For $b/n \geq 0.1$ it becomes more convenient – see below – to use our succinct bitvector, so we considered it more useful to fit memory usage in $b \in [0, 0.1]$. In any case – see plot 1 – the inferred model fits the experimental data well in the (wider) interval $b/n \in [0, 0.7]$.

[5] We estimated the impact of memory fragmentation by comparing RSS and allocated memory, after subtracting from RSS the estimated weight – approximately 3.4 MB – of loaded C++ libraries.

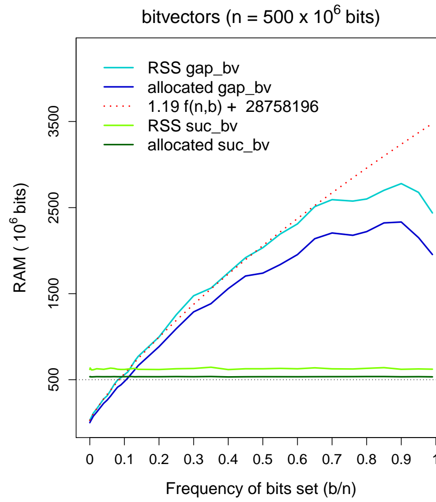


Figure 1 Memory occupancy of DYNAMIC's bitvectors. We show the growth of function $f(n,b) = b(\log(n/b) + \log\log b + \log n/\log b)$ opportunely scaled to take into account memory fragmentation and the weight of loaded C++ libraries.

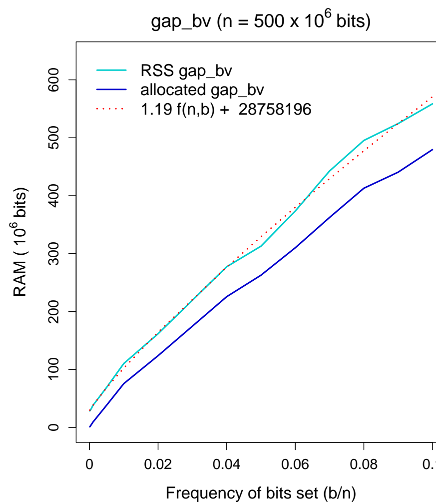


Figure 2 Memory occupancy of DYNAMIC's gap-encoded bitvector in the interval $b/n \in [0, 0.1]$ (for $b/n > 0.1$ the succinct bitvector is more space-efficient then the gap-encoded one). The picture shows that allocated memory closely follows our theoretical prediction (function $f(n,b)$).

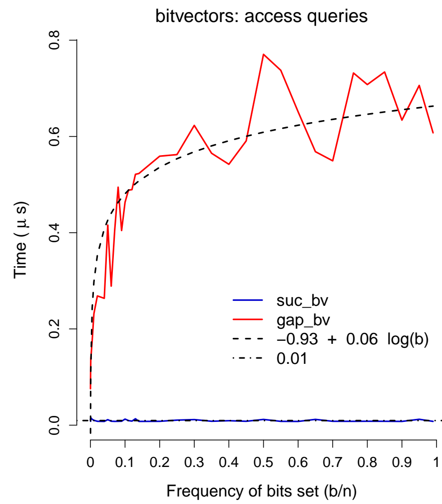


Figure 3 Running times of our bitvectors on `access` queries. Bitvectors' size is $n = 5 \times 10^8$ bits.

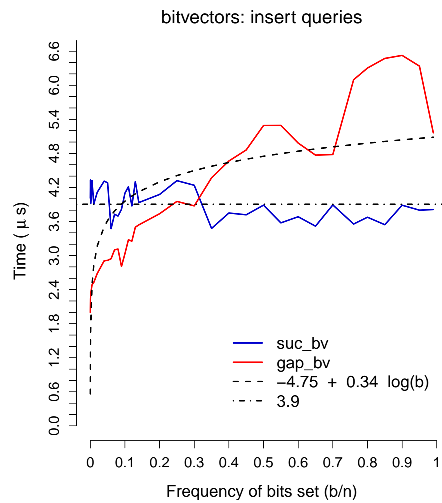


Figure 4 Running times of our bitvectors on `insert` queries. Bitvectors' size is $n = 5 \times 10^8$ bits.

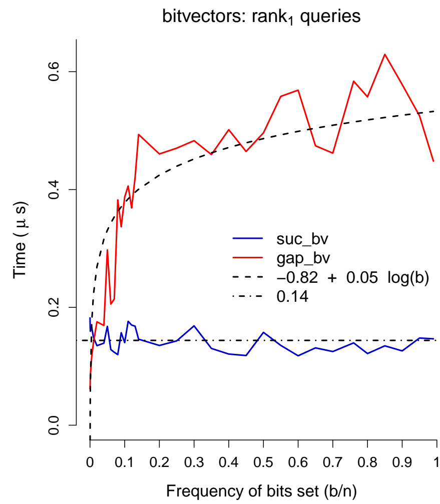


Figure 5 Running times of our bitvectors on $\texttt{rank}_1$ queries. Bitvectors' size is $n = 5 \times 10^8$ bits.

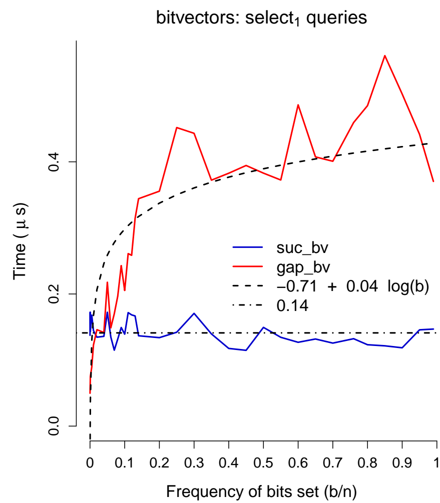


Figure 6 Running times of our bitvectors on $\texttt{select}_1$ queries.

3.2 An Application: Space-Efficient Compression Algorithms

We used DYNAMIC to implement five recently-published algorithms [26, 28, 27] computing the Burrows-Wheeler transform [3] (BWT) and the Lempel-Ziv 77 factorization [35] (LZ77) within compressed working space: cw-bwt [27] builds a BWT within $n(H_k + 1) + o(n \log \sigma)$ bits of working space by breaking it in contexts and encoding each context with a zero-order compressed string; rle-bwt builds the BWT within $\Theta(r)$ words of working space using the structure of Theorem 6; h0-lz77 [28] computes LZ77 online within $n(H_0 + 2) + o(n \log \sigma)$ bits using a dynamic zero-order compressed FM index; rle-lz77-1 and rle-lz77-2 [26] build LZ77 within $\Theta(r)$ words of space by employing a run-length encoded BWT augmented with a suffix array sampling based on BWT equal-letter runs and LZ77 factors, respectively. Implementations of these algorithms can be found within the DYNAMIC library [8]. We compared running times and working space of our algorithms against those of less space-efficient (but faster) state-of-the-art tools solving the same problems. BWT construction tools: se-sais [1, 12] ($\Theta(n)$ Bytes of working space), divsufsort [23, 12] ($\Theta(n)$ words), bwte [9] (constant user-defined working space; we always used 256 MB), dbwt [6] ($\Theta(n)$ Bytes). LZ77 factorization tools: isa6r [16, 19] ($\Theta(n)$ words), kkp1s [15, 19] ($\Theta(n)$ words), lzscan [14, 19] ($\Theta(n)$ Bytes). We generated two highly repetitive text collections by downloading all versions of the *Boost* library (github.com/boostorg/boost) and all versions of the English *Einstein*'s Wikipedia page (en.wikipedia.org/wiki/Albert_Einstein). Both datasets were truncated to $5 \cdot 10^8$ Bytes to limit RAM usage of the and computation times of the tested tools. The sizes of the 7-Zip-compressed datasets (www.7-zip.org) were 120 KB (Boost) and 810 KB (Einstein). The datasets can be found within the DYNAMIC library [8] (folder /datasets/). RAM usage and running times of the tools were measured using the executable /usr/bin/time.

In Figure 7 we report our results. Solid and a dashed horizontal lines show the datasets' sizes before and after compression with 7-Zip, respectively. Our tools are highlighted in red. We can infer some general trends from the plots. Our tools use always less space than the plain text, and from one to three orders of magnitude more space than the 7-Zip-compressed text. h0-lz77 and cw-bwt (entropy compression) always have working space very close to (and always smaller than) the plain text, with cw-bwt (k-th order compression) being more space-efficient than h0-lz77 (0-order compression). On the other hand, tools using a run-length compressed BWT – rle-bwt, rle-lz77-1, and rle-lz77-2 – are up to two orders of magnitude more space-efficient than h0-lz77 and cw-bwt in most of the cases. This is a consequence of the fact that run-length encoding of the BWT is particularly effective in compressing repetitive datasets. bwte represents a good trade-off in both running times and working space between tools working in compressed and uncompressed working space. kkp1s is the fastest tool, but uses a working space that is one order of magnitude larger than the uncompressed text and three orders of magnitude larger than that of rle-bwt, rle-lz77-1, and rle-lz77-2. As predicted by theory, tools working in compact working space (lzscan, se-sais, dbwt) use always slightly more space than the uncompressed text, and one order of magnitude less space than tools working in $\mathcal{O}(n)$ words. To conclude, the plots show that the price to pay for using complex dynamic data structures is high running times: our tools are up to three orders of magnitude slower than tools working in $\Theta(n)$ words of space. This is mainly due to the large number of insert operations – one per text character – performed by our algorithms to build the dynamic FM indexes.

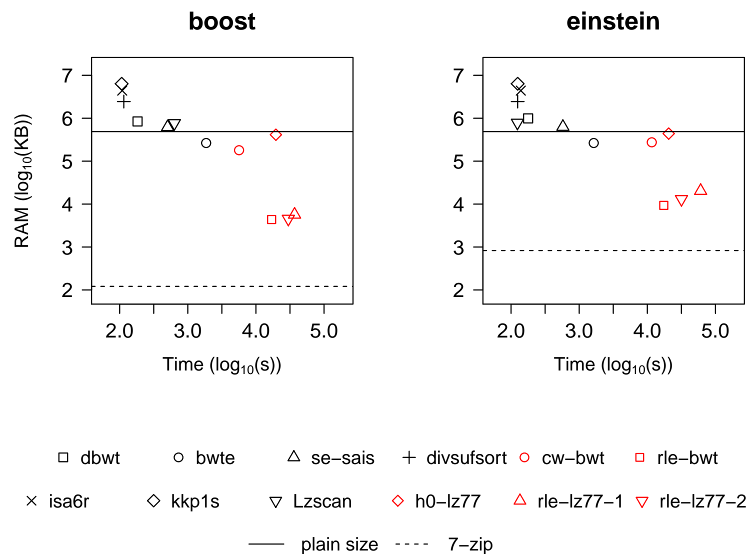


Figure 7 BWT and LZ77 compression algorithms. In red: tools implemented using DYNAMIC. Solid/dashed lines: space of the input files before and after 7-Zip compression, respectively.

References

1 Timo Beller, Maike Zwerger, Simon Gog, and Enno Ohlebusch. Space-Efficient Construction of the Burrows-Wheeler Transform. In *String Processing and Information Retrieval*, pages 5–16. Springer, 2013.

2 Daniel K. Blandford and Guy E. Blelloch. Compact representations of ordered sets. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 11–19. Society for Industrial and Applied Mathematics, 2004.

3 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm, 1994.

4 Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms (TALG)*, 3(2):21, 2007.

5 Joshimar Cordova and Gonzalo Navarro. Practical dynamic entropy-compressed bitvectors with applications. In *International Symposium on Experimental Algorithms*, pages 105–117. Springer, 2016.

6 dbwt: direct construction of the BWT. `http://researchmap.jp/muuw41s7s-1587/#_1587`. Accessed: 2016-11-17.

7 ds-vector: C++ library for dynamic succinct vector. `https://code.google.com/archive/p/ds-vector/`. Accessed: 2016-11-17.

8 DYNAMIC: dynamic succinct/compressed data structures library. `https://github.com/xxsds/DYNAMIC`. Accessed: 2017-01-22.

9 Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

11 bitvector: succinct dynamic bitvector implementation. `https://github.com/nicola-gigante/bitvector`. Accessed: 2016-11-17.

12 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. `doi:10.1007/978-3-319-07959-2_28`.

13 Roberto Grossi, Rajeev Raman, Satti Srinivasa Rao, and Rossano Venturini. Dynamic compressed strings with random access. In *International Colloquium on Automata, Languages, and Programming*, pages 504–515. Springer, 2013.
14 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Experimental Algorithms*, pages 139–150. Springer, 2013.
15 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching*. Springer, 2013.
16 Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 103–112. Society for Industrial and Applied Mathematics, 2013.
17 Patrick Klitzke and Patrick K. Nicholson. A general framework for dynamic succinct and compressed data structures. *Proceedings of the 18th ALENEX*, pages 160–173, 2016.
18 libcds: compact data structures library. `https://github.com/fclaude/libcds`. Accessed: 2016-11-17.
19 LZ77 factorization algorithms. `https://www.cs.helsinki.fi/group/pads/lz77.html`. Accessed: 2016-05-20.
20 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):32, 2008.
21 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. of Computational Biology*, 17(3):281–308, 2010.
22 Memoria: C++14 framework providing general purpose dynamic data structures. `https://bitbucket.org/vsmirnov/memoria/wiki/Home`. Accessed: 2016-11-17.
23 Y. Mori. Short description of improved two-stage suffix sorting algorithm, 2005.
24 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM Journal on Computing*, 43(5):1781–1806, 2014.
25 Pizza&Chili corpus. `http://pizzachili.dcc.uchile.cl`. Accessed: 2016-07-25.
26 A. Policriti and N. Prezza. Computing LZ77 in Run-Compressed Space. In *2016 Data Compression Conference (DCC)*, pages 23–32, March 2016. `doi:10.1109/DCC.2016.30`.
27 Alberto Policriti, Nicola Gigante, and Nicola Prezza. Average linear time and compressed space construction of the Burrows-Wheeler transform. In *International Conference on Language and Automata Theory and Applications*, pages 587–598. Springer, 2015.
28 Alberto Policriti and Nicola Prezza. Fast online Lempel-Ziv factorization in compressed space. In *International Symposium on String Processing and Information Retrieval*, pages 13–20. Springer, 2015.
29 Andreas Poyias, Simon J Puglisi, and Rajeev Raman. Compact Dynamic Rewritable (CDRW) Arrays. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 109–119. SIAM, 2017.
30 Andreas Poyias and Rajeev Raman. Improved practical compact dynamic tries. In *International Symposium on String Processing and Information Retrieval*, pages 324–336. Springer, 2015.
31 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Workshop on Algorithms and Data Structures*, pages 426–437. Springer, 2001.
32 Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *String Processing and Information Retrieval*, pages 164–175. Springer, 2009.
33 succinct library. `https://github.com/ot/succinct`. Accessed: 2016-11-17.
34 sux library. `http://sux.di.unimi.it/`. Accessed: 2016-11-17.
35 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Practical Range Minimum Queries Revisited

Niklas Baumstark[1], Simon Gog[2], Tobias Heuer[3], and Julian Labeit[4]

1 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`niklas.baumstark@student.kit.edu`
2 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`simon.gog@kit.edu`
3 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`tobias.heuer@student.kit.edu`
4 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`julian.labeit@student.kit.edu`

Abstract

Finding the position of the minimal element in a subarray $A[i..j]$ of an array A of size n is a fundamental operation in many applications. In 2011, Fischer and Heun presented the first index of size $2n + o(n)$ bits which answers the operation in constant time for any subarray. The index can be computed in linear time and queries can be answered without consulting the original array. The most recent and currently fastest practical index is due to Ferrada and Navarro (DCC'16). It reduces the range minimum query (RMQ) to more fundamental and well studied queries on binary vectors, namely rank and select, and a RMQ query on an array of sublinear size derived from A. A *range min-max tree* is employed to solve this recursive RMQ call. In this paper, we review their practical design and suggest a series of changes which result in consistently faster query times. Specifically, we provide a customized select implementation, switch to two levels of recursion, and use the *sparse table* solution for the recursion base case instead of a range min-max tree.

We provide an extensive empirical evaluation of our new implementation and also compare it to the state of the art. Our experimental study shows that our proposal significantly outperforms the previous solutions on established benchmarks (up to a factor of three) and furthermore accelerates real world applications such as traversing a succinct tree or listing all distinct elements in an interval of an array.

1998 ACM Subject Classification E.1 Data Structures, E.4 Coding and Information Theory

Keywords and phrases Succinct Data Structures, Range Minimum Queries, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.12

1 Introduction

Index data structures are computed once for a given input – for instance a document collection or a set of points – and can then be used to answer queries efficiently, without scanning the whole data set again. Hence, a query is reduced to operations whose running time is sublinear in the size of the original input. In the era of Big Data it is not surprising that index structures form the backbone of many search application, such as pattern matching in strings or range queries on point sets. The drawback of traditional index structures is that they are usually larger than the original data and have to reside in main memory to facilitate fast queries. An example are pointer-based search trees. This motivates the development of space-efficient index structures which often are not only substantially smaller than traditional

© Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit;
licensed under Creative Commons License CC-BY

indexes but can also answer queries without access to the original data. In a seminal paper, Jacobson showed in 1989 how an arbitrary tree of n nodes can be represented in $2n + o(n)$ bits while traversal operations such as finding the parent and children of a node are still supported in constant or logarithmic time [14]. The tree is represented as a vector over $\{0, 1\}$ and navigation is reduced to two operations: $\text{RANK}_1(i, B)$ counts the number of bits set in the prefix of size i of the binary vector B and $\text{SELECT}_1(j, B)$ returns the position of the j-th set bit in B. Today, there exist many other space-efficient counterparts of classic structures which are based on these two fundamental operations; compressed text indexes such as the FM-index are probably the most prominent example [8]. More examples can be found in Navarro's textbook [16].

Improving the performance of basic data structures in this area is of utmost importance as an improvement will often directly translate to faster text indexes or other complex structures [11]. In this paper, we develop a space-efficient data structure to solve range minimum queries, which occur as sub-problems in many different applications. To give a concrete example, they arise in information retrieval problems such as top-k completion [13] and general top-k document retrieval [16].

▶ **Definition 1.** Given an array $A[1..n]$ of n numbers. The *range minimum query* (RMQ) problem is to find an index structure that returns for any range $A[i..j]$ the position of the leftmost minimum. More formally, for any pair of positions $1 \leq i \leq j \leq n$,

$$\text{RMQ}_A(i, j) = \underset{i \leq k \leq j}{\arg\min} \langle A[k], k \rangle \, .$$

Two tuples $\langle a, b \rangle$ and $\langle c, d \rangle$ are compared lexicographically, i.e. $\langle a, b \rangle < \langle c, d \rangle \iff a < c \vee (a = c \wedge b < d)$.

Most solutions of the RMQ problem are based on the reduction to a restricted version of the problem. The restriction is that adjacent entries of the input array $A[1..n]$ only differ by ± 1. Such arrays can be represented as a bit vector B, where $B[1] = 1$ and $B[i] = 1$ if $A[i] - A[i-1] = +1$ and $B[i] = 0$ otherwise. An entry $A[i]$ can be reconstructed by $A[i] = \text{RANK}_1(i, B) - \text{RANK}_0(i, B) = 2 \cdot \text{RANK}_1(i, B) - i$. RANK_0 computes the number of zero bits in a prefix of a vector, analogously to RANK_1. This formula is also called $\text{EXCESS}(i, B)$ and we use it to define the restricted problem:

▶ **Definition 2.** Given a bit vector $B[1..n]$, the ± 1RMQ problem is to find an index structure that returns for any range $B[i..j]$ the position of the leftmost and rightmost excess minimum. More formally, for any pair of positions $1 \leq i \leq j \leq n$,

$$\text{RMQ}^{\pm}_B(i, j) = \underset{i \leq k \leq j}{\arg\min} \langle \text{EXCESS}(k, B), k \rangle \, ,$$

$$\text{RRMQ}^{\pm}_B(i, j) = \underset{i \leq k \leq j}{\arg\min} \langle \text{EXCESS}(k, B), -k \rangle \, .$$

The RMQ problem is well studied and various solutions have been proposed. The first optimal space index requires only $2n + o(n)$ bits and was invented by Fischer and Heun [9]. It can be built in linear time and answers each query in constant time without accessing the original array. Several authors implemented variants of the index [12, 11, 5]. Here, we briefly describe the general idea behind these solutions in order to highlight the contributions of this paper. Any range minimum query can be translated to a *lowest common ancestor query* (LCA) on the *Cartesian tree* of A. The tree can be stored in a succinct representation which uses a bitvector B of length $2n$. Mapping an array element of A to a node in the tree is

reduced to a SELECT operation. The LCA operation can be translated back into a ± 1RMQ query on the succinct tree representation. There are several options to solve the ± 1RMQ problem, and most practical implementations use the *range min-max tree* to do so [17, 1]. In a last step the result of the ± 1RMQ is mapped back to the corresponding position on A, via a RANK operation. Recently, Ferrada and Navarro [5] showed that with a specific tree representation no more than three basic operation calls ($2 \times$ SELECT, $1 \times$ RANK) are required on top of the ± 1RMQ call to answer a query[1]. Previous implementations in the SDSL [11] and SUCCINCT [12] library require the execution of another relatively expensive basic operation for an ancestor test. To decrease space usage, Ferrada and Navarro replaced the constant time index for SELECT by a binary search over the RANK index [6] or a combination of select dictionary and scanning [5]. While this negatively affects the overall query time, they show that their solution is still the fastest on a wide range of benchmarks. However, they also noted that the library implementations are still faster on real-world applications such as suffix tree traversal.

In this paper we suggest a series of improvements to Ferrada and Navarro's index. Specifically,

- We show that the knowledge about the height of the Cartesian tree can be used to accelerate SELECT. For trees with logarithmic height we get constant query time without using any extra space. This is a major improvement compared to other previous implementations.
- Navarro & Ferrada proposed two succinct tree representations (rightmost and leftmost path-mapped general tree). One can choose between the two options so as to minimize the height of the tree, but they require different basic structures (RANK$_1$/RANK$_0$ and rightmost/leftmost ± 1RMQ). We use a single mapping and simulate the other by reversing the input.
- We introduce an effective optimization for small query ranges. Replacing the second SELECT for the right border of the range by a local scan on the parentheses sequence significantly improves the performance on real world applications.
- We replace the traditionally used range min-max tree by a recursive call to our optimized solution and resort to the sparse table approach after a constant number of recursions.

Combining these optimizations we obtain an index that outperforms the existing implementations not only on established benchmarks but also on real-world applications. The remainder of the paper is organized as follows: In Section 2 we review the previous work and present the simplified framework of Ferrada and Navarro. In Section 3 we present our optimizations in detail and subsequently provide experimental evidence of their effectiveness in Section 4.

2 Previous Work

A straightforward constant query time solution to the RMQ problem is to precompute every possible query and store each answer into a lookup table of size $\mathcal{O}\left(n^2\right)$. However the memory requirement of this approach is prohibitively high. Bender & Farach-Colton [2] presented an elegant $\mathcal{O}\left(n \log n\right)$-space solution, which uses a sparse version of the naive lookup table. They precompute a table $M[1..n][1..\log n]$ with $M[i][j] = \text{RMQ}_A(i, i + 2^j - 1)$; i.e. for all queries with an interval size that is a power of two, the answers are stored. An arbitrary query

[1] We note that this simplified approach was also described by Davoodi et al. [4] in CACOON 2012.

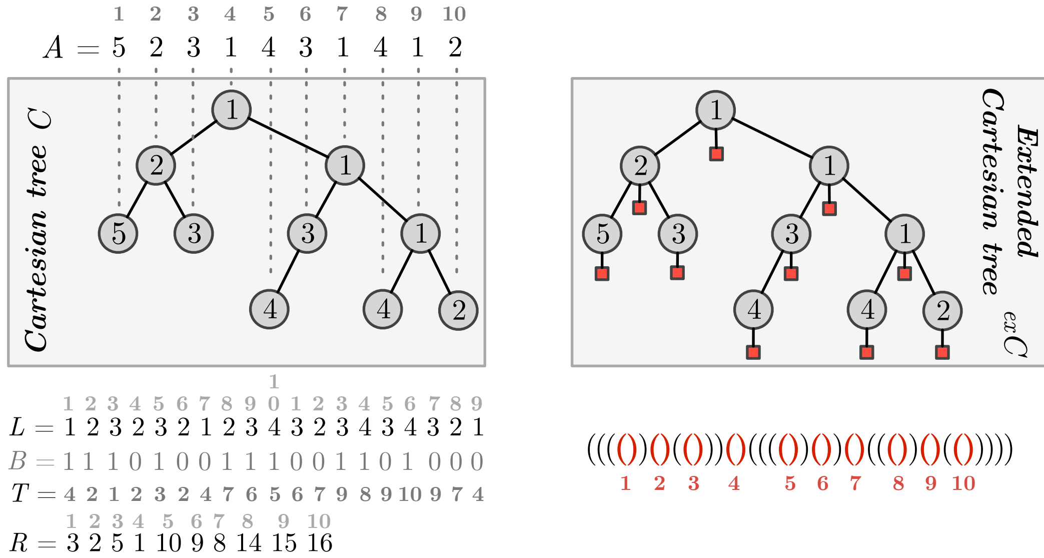


Figure 1 Example of a Cartesian and extended Cartesian tree. Note that we use the leftmost tie-breaking policy.

$\text{RMQ}(i,j)$ can be answered by first determining the maximal k with $2^k \leq j-i+1$. Then $A[M[i,k]]$ and $A[M[j-2^k+1,k]]$ are compared to determine the result. The method requires just four memory accesses (two to A and two to M), a comparison, and a *log*-calculation on integers which corresponds to counting leading zero bits[2]. We refer to this solution as SparseTable. More space-efficient RMQ solutions are based on the Cartesian tree introduced in [19].

▶ **Definition 3.** Given an array $A[1..n]$ of numbers. The *Cartesian tree* $\mathcal{C}$ is a binary tree which is recursively defined as follows: (1) If A is empty, then $\mathcal{C}(A)$ is the empty tree. (2) Otherwise, let $p = \arg\min_{1\leq i\leq |A|}\langle A[i], i\rangle$ be the position of the leftmost minima. The root of $\mathcal{C}(A)$ is element $A[p]$. Its left subtree is $\mathcal{C}(A[1..p-1])$ if $p > 1$ and its right subtree $\mathcal{C}(A[p+1..|A|])$ if $p < |A|$.

We denote as $\mathcal{C}_L/\mathcal{C}_R$ the Cartesian tree where leftmost/rightmost minima are selected in the case of ties, respectively.

Figure 1 shows the Cartesian tree for an example array. Note that one can map between array entries and nodes. The in-order index $\text{Inorder}(v)$ of a node v corresponds to the nodes index in A. Conversely, we define the mapping from an index to its node with Innode[3]. Gabow et al. observed that an RMQ query in A can be reduced to calculating the lowest common ancestor (LCA) in $\mathcal{C}(A)$ [10].

$$\text{RMQ}_A(i,j) = \text{Inorder}(\text{LCA}_{\mathcal{C}(A)}(\text{Innode}(i), \text{Innode}(j)))\,.$$

Berkman and Vishkin noted, that the problem of calculating the LCA can again be reduced to an RMQ query on an array L of the depths of the nodes in the *Euler tour* T through $\mathcal{C}(A)$ [3]. In Figure 1 arrays $T[1..2n]$ and $L[1..2n]$ contain the nodes (identified by their

[2] This operation is part of the instruction set of most modern CPUs.
[3] Note that we omit the mapping in cases where we can directly identify nodes by their in-order index.

in-order index) and their corresponding depth in the Euler tour. Note that L can be replaced by a bit vector B as $L[i] = 2 \cdot \text{RANK}_1(i, B) - i$). With an additional array $R[1..n]$ which contains the first occurrence of each node in the Euler tour LCA queries can be answered as follows:

$$\text{LCA}_{\mathcal{C}(A)}(\text{INNODE}(i), \text{INNODE}(j)) = \text{INNODE}(T[\text{RMQ}^{\pm}_{B}(R[i], R[j])]) \,.$$

Bender & Farach-Colton [2] solve the ± 1RMQ problem by partitioning B into blocks of size $s = \frac{1}{2}\log n$ and creating a SPARSETABLE structure over the $\frac{n}{s}$ block minima. As there can be at most $2^s = \sqrt{n}$ different block types, one can afford to store a lookup table for all $\mathcal{O}\left(2^s \cdot s^2\right)$ in-block RMQ queries. For an arbitrary range $[i..j]$, the query is divided into three sub-queries, including at most two in-block queries in the case where i and j are not block aligned and a SPARSETABLE query for the blocks with indexes in the interval $[\lceil \frac{i+1}{s} \rceil, \lfloor \frac{j-1}{s} \rfloor]$. The EXCESS values of the three positions are compared and the position of the leftmost minimum is returned. This solution requires a linear number of words and the space is dominated by the SPARSETABLE on the sequence of $\frac{2n}{\log n}$ block minima.

Sadakane [18] showed that the ± 1RMQ problem can be solved with just sublinearly many bits in addition to B by first dividing B into blocks of size $\log^3 n$ (SPARSETABLE requires $\mathcal{O}\left(\frac{n}{\log n}\right) \in o(n)$ bits), subdividing those into in sub-blocks of size $\frac{1}{2}\log n$ (SPARSETABLE in total again in $o(n)$ bits) and handling the inner blocks again with tabulation as above. He also observed that B can be transformed into a succinct representation – the *balanced parentheses sequence* (BP) of $\mathcal{C}$ – by replacing 1/0 with opening/closing parentheses and appending a closing parenthesis at the end.

BPs were originally introduced by Munro & Raman [15]. The $2n$-bits BP is defined by a depth-first traversal where an opening parenthesis is appended when arriving at a node v and a closing parenthesis is appended after processing v's subtree. Nodes in BP are either in DFS-preorder (if they are identified with their corresponding opening parenthesis) or in DFS-postorder. To use the presented RMQ framework, which is based on in-order, Sadakane introduced the *extended Cartesian tree ex*$\mathcal{C}$, which adds one pseudo-leaf per node (see Figure 1) [18]. These n leaves serve as in-order markers of the original nodes. Now INNODE and INORDER can be realized by SELECT and RANK on the parentheses pattern "`()`" (or "`10`" if interpreted as bits) and the query can be expressed as follows:

$$\text{RMQ}_A(i, j) = \text{RANK}_{\texttt{10}}(\text{RMQ}^{\pm}_{BP(ex\mathcal{C})}(\text{SELECT}_{\texttt{10}}(i), \text{SELECT}_{\texttt{10}}(j))) \,. \tag{1}$$

In 2016, Ferrada and Navarro [5] showed how the balanced parentheses representation can be directly applied to the Cartesian tree. They transform the binary Cartesian tree $\mathcal{C}$ into a leftmost path-mapped general Cartesian tree $\mathcal{C}^L$ (see Figure 2) using a known mapping [15]. A new pseudo-root is introduced and the node from the leftmost path in the binary tree (from the leaf to root) are attached to the new root. This process is applied recursively on the subtrees of the former leftmost path. In $\mathcal{C}^L$ the node with pre-order index $i+1$ corresponds to the node with in-order index i in $\mathcal{C}$ (the $+1$ is due to the added pseudo-root). INNODE and INORDER can by directly realized by SELECT and RANK on the opening parentheses and the RMQ expressed as follows:

$$\text{RMQ}_A(i, j) = \text{RANK}_{\texttt{1}}(\text{RRMQ}^{\pm}_{BP(\mathcal{C}^L)}(\text{SELECT}_{\texttt{1}}(i+1) - 1, \text{SELECT}_{\texttt{1}}(j+1))) \,. \tag{2}$$

Note that the ± 1RMQ has to return the rightmost minimum. Alternatively, they provide the formula for the rightmost path mapping ($\mathcal{C}^R$); see Figure 2. In this case pre-order is

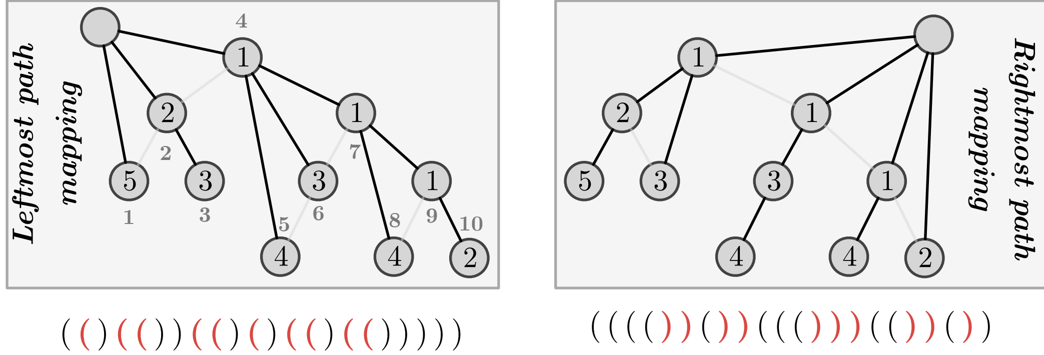


Figure 2 Transformation of a binary Cartesian tree into a general tree. We show the leftmost path mapping ($\mathcal{C}^L$) and rightmost path mapping ($\mathcal{C}^R$) for the example of Figure 1.

replaced by post-order, $\textsc{select}_1/\textsc{rank}_1$ by $\textsc{select}_0/\textsc{rank}_0$ and the ± 1RMQ has to return the *leftmost* minimum.

$$\mathrm{RMQ}_A(i,j) = \textsc{rank}_0(\mathrm{RMQ}^{\pm}_{BP(\mathcal{C}^R)}(\textsc{select}_0(i), \textsc{select}_0(j)))\,. \tag{3}$$

Ferrada and Navarro already noted the possibility of constructing indexes based on both mappings and choosing the more attractive one, e.g. the one which minimizes the height of the tree. In Figure 2 $\mathcal{C}^L$ has a depth of five, while $\mathcal{C}^R$ has a depth of four. This observation acts as a starting point of our work.

3 An Optimized Recursive Solution

Our first optimization considers the $\textsc{select}_1$ operation, which is used to find the position of node v with $\textsc{Preorder}(v) = i$ when answering range minimum queries using Equation 2. The result of $\textsc{select}$ can be directly determined if $\textsc{depth}(v)$ is known:

▶ **Lemma 4.** *Let T be a rooted ordered tree and BP be the balanced parentheses sequence of T. The position of the representing opening parenthesis in B of a node $v \in T$ with $\textsc{Preorder}(v) = i$ is $\textsc{select}_1(i, BP) = 2i - \textsc{depth}(v) - 1$, where $\textsc{depth}(v) \geq 0$ denotes the distance of v to the root node.*

Proof. By construction the opening parenthesis of v is appended to BP after processing the $i-1$ nodes with smaller pre-order IDs. The $i - 1 - \textsc{depth}(v)$ nodes which are not ancestors of v were fully processed and we have written a parentheses pair for each of them. The $\textsc{depth}(v)$ ancestors are not yet fully processed as v is a node in their subtree. So we have written only one parenthesis per ancestor. In total we output v's opening parenthesis at position $2 \cdot (i - 1 - \textsc{depth}(v)) + \textsc{depth}(v) + 1 = 2i - \textsc{depth}(v) - 1$ in an 1-index based sequence. ◀

The answer of every $\textsc{select}_1$ query can therefore be estimated using the depth of T or the maximal excess in BP More precisely the answer must be within the interval $BP[2i - \textsc{max_excess}(BP) - 1..2i - 1]$, where $\textsc{max_excess}(BP) := \max_{0 \leq i \leq |BP|} \textsc{excess}(i, BP)$. We note that for trees of logarithmic depth we can directly calculate the correct 64-bit word containing the answer without using any extra space. To guarantee constant running time for larger tree sizes, we can employ a sampling scheme similar which uses $o(n)$ space and is very similar to the traditional solution for $\textsc{select}$.

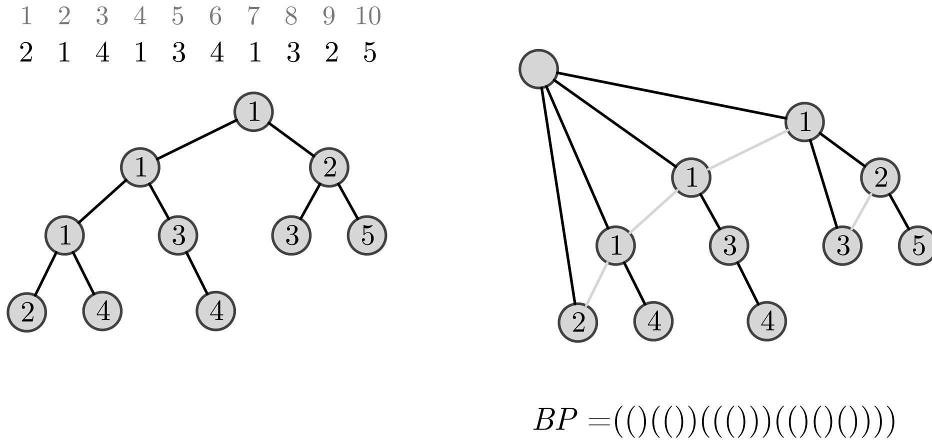


Figure 3 Left: Cartesian tree $\mathcal{C}_R(A[n..1])$ built over $A[n..1]$ (the reverse array of the example in Figure 1; the rightmost tie-breaking policy is used. Right: Leftmost-path general tree $\mathcal{C}_R^L(A[n..1])$ of $\mathcal{C}_R(A[n..1])$. Note that this is the mirrored tree of $\mathcal{C}_L^R(A[1..n])$ in Figure 2 (right).

Next we show that it is not actually necessary to implement the rightmost mapping and its underlying primitives as it can be simulated using the leftmost mapping on the reversed input and a changed tie-breaking policy.

▶ **Theorem 5.** *Let $A[1..n]$ be an array of integers and $\overleftarrow{A} = A[n..1]$ the array in reverse order. Then $\mathcal{C}_L^R(A)$ is isomorphic to $\mathcal{C}_R^L(\overleftarrow{A})$.*

Proof. First we show that reversing the input and changing the tie braking policy yields the same Cartesian tree, hence $\mathcal{C}_L(A)$ is isomorphic to $\mathcal{C}_R(\overleftarrow{A})$. The statement is trivially true for $n = 1$. For $n > 1$ the root of $\mathcal{C}_L(A)$ is the leftmost minimum $m = A[p]$. As $\overleftarrow{A}$ contains the same values as A, m also has to be the minimum in $\overleftarrow{A}$. The mirrored position $p' = n+1-p$ contains m and has to be the rightmost minimum in $\overleftarrow{A}$. Otherwise, there would be a $p'' > p'$ with $\overleftarrow{A}[p''] = m$ which is mapped to a position $q = n+1-(n+1-p) < p$ with $A[q] = m$. This contradicts the assumption that $A[p]$ is the leftmost minimum. The left child of the root of $\mathcal{C}_L(A)$ is $\mathcal{C}_L(A[1..p-1])$ and the right child of the root of $\mathcal{C}_R(\overleftarrow{A})$ is $\mathcal{C}_R(\overleftarrow{A}[p'+1..n])$. By definition $\overleftarrow{A}[p'+1..n]$ is the reverse array of $A[1..p-1]$. By induction $\mathcal{C}_L(A[1..p-1])$ is isomorphic to $\mathcal{C}_R(\overleftarrow{A}[p'+1..n])$. The same argument can be apply to $\overleftarrow{A}[1..p'-1]$ and $A[p+1..n]$. Thus $\mathcal{C}_L(A)$ is isomorphic to $\mathcal{C}_R(\overleftarrow{A})$.

Further we observe that $\mathcal{C}_L(A)$ is the mirrored version of $\mathcal{C}_R(\overleftarrow{A})$. I.e. the rightmost path in $\mathcal{C}_L(A)$ corresponds to the leftmost path in $\mathcal{C}_R(\overleftarrow{A})$. Therefore, the leftmost path mapped tree $\mathcal{C}_R^L(\overleftarrow{A})$ of $\mathcal{C}_R(\overleftarrow{A})$ is the mirrored version of the rightmost path mapped tree $\mathcal{C}_L^R(A)$ of $\mathcal{C}_L(A)$. Hence $\mathcal{C}_L^R(A)$ is isomorphic to $\mathcal{C}_R^L(\overleftarrow{A})$. ◀

The right tree in Figure 2 and the right tree in Figure 3 show both trees, $\mathcal{C}_L^R(A)$ and $\mathcal{C}_R^L(\overleftarrow{A})$, for our running example. A query $\mathrm{RMQ}_A(i,j)$ can be answered with $\mathcal{C}_R^L(\overleftarrow{A})$ as follows. The query range $[i,j]$ is mirrored $[\mu(j),\mu(i)]$, with $\mu(x) = n+1-x$, and we get the position p of the rightmost minimum in $\overleftarrow{A}[\mu(j),\mu(i)]$ as $\mathcal{C}_R^L$ breaks ties with rightmost policy. The mirrored position $\mu(p)$ of p in turn is the leftmost minimum in $A[i,j]$. This observation helps to simplify the query algorithm, as it does not need to support both rightmost and the leftmost mapping.

The technique of reversing the input sequence and adjusting the query range can also be used to implement ±1RRMQ and we therefore get a recursive algorithm. Remember that Bender &Farach-Colton divided $BP(\mathcal{C})$ into blocks of fixed size s and built the structure again over the sequence E of block minima. The recursive structure built for E again profits from our proposed optimizations in Lemma 4 and of Theorem 5. The recursion is terminated when the length of E is in $o(\frac{n}{\log^2 n})$ and the SparseTable structure can be applied.

Algorithm 1 summarizes the construction of our RMQ-index. The recursion base case in Line 3 constructs SparseTable. For the remaining Λ levels, first the leftmost-path mapped Cartesian tree with leftmost tie breaking policy $\mathcal{C}_L^L$ over A and the leftmost-path mapped Cartesian tree with rightmost tie breaking policy $\mathcal{C}_R^L$ over the reverse of A is built. The tree of minimal depth is selected and its balanced parentheses sequence stored in BP_λ along with a flag indicating whether A was reversed; see Lines 5–8. Next, BP is partitioned into blocks of size s_λ and two new arrays of size $\frac{n}{s_\lambda}$ are generated in linear time by iterating over BP: Array I_λ and I_λ contain for each block of BP the position respectively the excess-value of the rightmost element with minimal excess. In the pseudo-code, the entries in I_λ are considered as absolute positions in A_λ. In practice these values are stored relative to the start index of each block. So only $\log s_\lambda$ bits per element are required. The second array E_λ contains the excess-value for each entry in I_λ. The entries in E_λ are stored as absolute values, each taking $\log \text{MAX_EXCESS}(BP)$ bits. In Line 11 we recursively index E. Note that we index the *reverse* $\overleftarrow{E}$ of E in the recursive call. We will see that this approach results in a very simple query algorithm.

The time complexity of Algorithm 1 is linear in the original input size n for an appropriate choice of Λ and s. For $\Lambda = 3$ and $s = [\log n, \log n, \log n]$ the SparseTable structure in Line 3 is built in the fourth recursive level for an array of size $n''' = \frac{n}{\log^3 n}$; i.e. as SparseTable is constructed in $\mathcal{O}(n''' \log n''')$ this step takes $o(n)$ time. It is easy to see the all remaining steps up to and including the third recursion level take linear time.

The space complexity for this choice is $2n + o(n)$ bits if E_λ is represented implicitly[4]. The $2n$-bit term is due to BP_1 and storing the relative values of I_1 we get additional $\mathcal{O}\left(\frac{n \log \log n}{\log n}\right) = o(n)$ bits of space for the first level. For deeper recursion levels the input is sublinear in n and we can therefore store both BP_λ and I_λ in sublinear space. Finally, SparseTable is in $o(n)$ as the length of the input was reduced to $n''' = \frac{n}{\log^3 n}$ and SparseTable takes $\mathcal{O}\left(n''' \log^2 n'''\right)$ bits.

We have split the query implementation in two parts. The general leftmost RMQ query (see Algorithm 2) and the rightmost ±1RMQ query (see Algorithm 3). We start by explaining Algorithm 2. The recusion base case in Line 3 is easily solved by the precomputed SparseTable. Otherwise we follow Equation 2 by using select to map the query interval $[i, j]$ to position in BP_λ (see Line 6 and 9), solving the ±1RRMQ and mapping the corresponding position back to the original array via rank (see Line 10). Note that handling the tree height minimization only required minor adjustments: First, we introduce a function $\mu(x) = x + rev_\lambda \cdot (n+1-x)$ which maps a position x in the original array to the corresponding position in the preprocessed array. Second, we swap the left and right bound of the given query range for cases where the array was reversed during preprocessing (see Line 4 and 5). This ensures that $\mu(i+1) \le \mu(j+1)$ and the range $[\ell, r]$ for the recursive call has a positive size.

[4] This can be achieve by a rank structure over BP_λ.

Algorithm 1 Recursive construction of a Λ-level RMQ-index with leftmost tie-breaking policy for an array A with block sizes s. The λ parameter corresponds to the current recursion level. On level λ the data structure stores the parentheses sequence BP_λ, a flag rev_λ which indicates whether the sequence was reversed, and two arrays (I_λ, E_λ) which contain for each block of BP_λ the position respectively the EXCESS-value of its rightmost element with minimal EXCESS.

```
1: procedure PREPROCESSING(A[1..n], λ, Λ, s = [s_1, ..., s_Λ])
2:     if λ > Λ then
3:         construct SPARSETABLE over A
4:     else
5:         if depth(C_L^L(A[1..n])) ≤ depth(C_R^L(A[n..1])) then
6:             ⟨BP_λ, rev_λ⟩ ← ⟨C_L^L(A[1..n]), 0⟩
7:         else
8:             ⟨BP_λ, rev_λ⟩ ← ⟨C_R^L(A[n..1]), 1⟩

9:         I_λ ← [±1RRMQ_{BP_λ}((i − 1)s_λ + 1, i · s_λ) | 1 ≤ i ≤ (2n+2)/s_λ]
10:        E_λ ← [EXCESS(i, BP_λ) | i ∈ I_λ]

11:        PREPROCESSING(E_λ[n..1], λ + 1, Λ, s)        ▷ Recursive construction
```

In the experimental part we will observe that most of the query time is spent on the two SELECT operations in Line 6 and 9). This motivates the optimization in Line 7. For small ranges ($j - i < \log n$) we scan a constant number of words for the index and excess of the rightmost minimum. On success ($\neq \perp$), i.e. we reached the j-th opening parenthesis during the scan, we output the result and avoid the second select call in Line 9.

In Algorithm 3 we finally outline our ±1RRMQ implementation. The basic concept follows the idea of Bender & Farach-Colton (see Section 2). In Line 2 we determine the range of blocks $[\ell', r']$ which intersects the query interval $[\ell, r]$. In the next line, we recursively determine the position of the rightmost minimum excess value in the blocks, which are fully contained inside the query interval $[\ell, r]$. We mirror the range bounds as well as the result, since we built the RMQ over the reversed array $\overleftarrow{E}$ of E (see Line 11 in Algorithm 1). Note that the recursive call returns the leftmost minimum of the reversed array, which corresponds to the desired rightmost minimum in the non-reversed array. Next, we obtain the position and EXCESS-value of the rightmost minimum in the two fringe blocks ℓ' and r' by accessing the corresponding entries of I and E. If an obtained position is outside the query range $[\ell, r]$ and the corresponding EXCESS-value smaller than the EXCESS-value obtained from the recursive call, we scan the fringe block to identify the position and EXCESS-value of the rightmost minimum inside the query range. We note, that such a strategy to avoid local scans was similarly suggested by Ferrada and Navarro [5] in the context of range min-max trees.

It is easy to see that the time complexity of a RMQ query is constant. There are a constant number of recursive calls and all basic operations, except SCAN, require only constant time. Note that SCAN can be implemented in constant time for blocks of size $\frac{1}{2}\log n$ by employing lookup tables of sublinear size. In the next section, we will see that scanning a block of a reasonable size, e.g. a constant number of cache lines, will not dominate the practical query time.

Algorithm 2 Recursive query implementation on a Λ-level RMQ-index with leftmost tie-breaking policy. We use two helper functions in the code: An array position x is mapped to the corresponding position in the reversed or non-reversed array by $\mu(x) = x + rev_\lambda \cdot (n+1-2x)$. Method $\textsc{Scan}(BP)$ returns a (EXCESS-value,position)-pair of the rightmost element with minimal EXCESS at or to the left of the j-th opening parentheses. In case the j-th opening parentheis is not located in the block $\perp$ is returned.

```
1: procedure RMQ(i, j, λ, Λ)
2:     if λ > Λ then
3:         return SPARSETABLE(i, j)

4:     if rev_λ = 1 then                 ▷ If C_R^L was built on the reversed sequence on level λ
5:         ⟨i, j⟩ ← ⟨j, i⟩                                        ▷ swap left and right bound

6:     ℓ ← SELECT_1(μ(i + 1), BP_λ) − 1
7:     if j − i < log n and (⟨e, p⟩ ← SCAN(BP_λ[ℓ..ℓ + 2 log n])) ≠ ⊥ then
8:         return μ(RANK_1(p, BP_λ))
9:     r ← SELECT_1(μ(j + 1), BP_λ)

10:    return μ (RANK_1(±1RRMQ(l, r, λ, Λ), BP_λ))                      ▷ Cf. Equation 2
```

4 Experimental Evaluation

We created a generic C++ implementation of the proposed RMQ-index. We refer to it as NewRMQ[5] and compare it to the follows baselines: the two library implementations Sdsl[6] [11] and Succinct[7] [12] and the code of Ferrada and Navarro [5] (F&N'16[8]).

The experiments were executed on a single core of a machine equipped with four Intel Xeon E5-4640 processors, with a combined number of 32 cores and 64 hyper-threads, and 512 GiB of memory. All programs were compiled using GCC 4.8.4 with optimizations turned on.

Following the methodology in [5] we generated three variants of artificial inputs. (1) Random inputs RANDOM: Values were drawn uniformly at random from the range $[1, n]$. (2) Pseudo-increasing inputs INC-δ: For a given δ, entry $A[i]$ was chosen at random in $[i - \delta, i + delta]$. (3) Pseudo-decreasing inputs DEC-δ: For a given δ, entry $A[i]$ was chosen at random in $[n - i - \delta, n - i + \delta]$. We varied input lengths ($n = 10^x, x \in \mathbb{N}_+$) and for each variant and generated 10^4 random queries $[i, j]$ for each range size $j - i + 1 \in \{10^1, .., 10^{x-1}\}$.

In an initial experiment we explored the effect of varying the block size s_λ and number of recursion levels Λ in NewRMQ on RANDOM. While we tested a large range of block sizes and recursion levels we restrict our presentation to the most promising parameters. Specifically, $s \in \{1024, 2048, 4096\}$, which corresponds to 2, 4, and 8 cache lines, and $\Lambda \in \{1, 2, 3\}$. Figure 4 depicts query times and Table 2 index space. In the following we excluded all version with $s \in \{2048, 4069\}$ since the query time for median sized intervals was much worse than for $s = 1024$. We also excluded the $\Lambda = 1$ variants due to their larger memory overhead

[5] Our code is available at `https://github.com/kittobi1992/rmq-experiments`.
[6] Available at `https://github.com/simongog/sdsl-lite` (Accessed at 20.12.2016).
[7] Available at `https://github.com/ot/succinct` (Accessed at 20.12.2016).
[8] Available at `https://github.com/hferrada/rmq` (Accessed at 20.12.2016).

Algorithm 3 Rightmost ±1RMQ query implementation on a Λ-levelRMQ-index.

1: **procedure** $\pm$1RRMQ$(\ell, r, \lambda, \Lambda)$
2: $\quad\langle \ell', r' \rangle \leftarrow \langle \left\lceil \frac{\ell+1}{s_\lambda} \right\rceil - 1, \left\lfloor \frac{r}{s_\lambda} \right\rfloor + 1 \rangle$ ▷ Leftmost/rightmost covered blocks
3: $\quad p' \leftarrow |E_\lambda| + 1 - \text{RMQ}(|E_\lambda| + 1 - (r'-1), |E_\lambda| + 1 - (\ell'+1), \lambda + 1, \Lambda)$ ▷ Recurse
4: $\quad\langle \langle p_{\ell'}, e_{\ell'} \rangle, \langle p_{r'}, e_{r'} \rangle \rangle \leftarrow \langle \langle I_\lambda[\ell'], E_\lambda[\ell'] \rangle, \langle I_\lambda[r'], E_\lambda[r'] \rangle \rangle$
5: $\quad$**if** $e_{\ell'} < E_\lambda[p'] \wedge p_{\ell'} < \ell$ **then** ▷ Try to avoid SCAN of leftmost block.
6: $\quad\quad\langle p_{\ell'}, e_{\ell'} \rangle \leftarrow \text{SCAN}(BP_\lambda[\ell..(\ell'+1)s_\lambda])$
7: $\quad$**if** $e_{r'} < E_\lambda[p'] \wedge p_{r'} > r$ **then** ▷ Try to avoid SCAN of rightmost block.
8: $\quad\quad\langle p_{r'}, e_{r'} \rangle \leftarrow \text{SCAN}(BP_\lambda[(r'-1)s_\lambda..r])$
9: $\quad\langle e, -p \rangle \leftarrow \min\{\langle e_{\ell'}, -p_{\ell'} \rangle, \langle E_\lambda[p'], -p' \rangle, \langle e_{r'}, -p_{r'} \rangle\}$
10: $\quad$**return** p

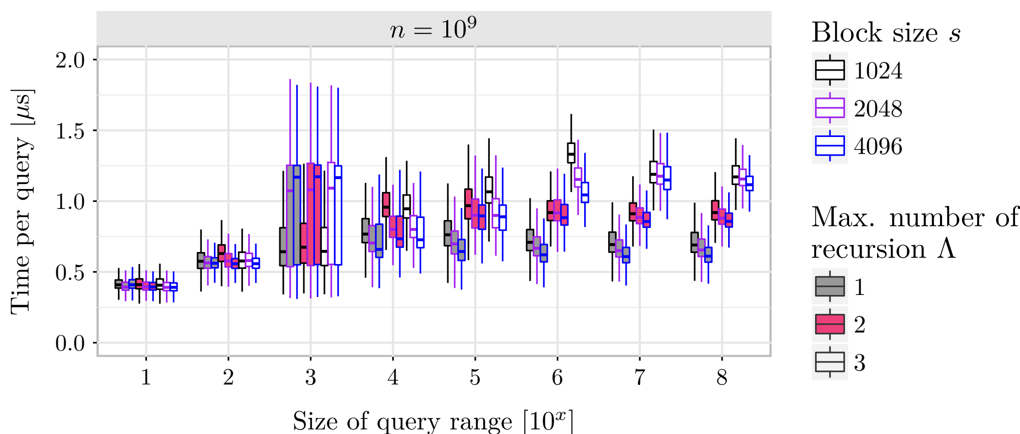


Figure 4 Query time distribution for the recursive RMQ-index on input RANDOM.

and the $\Lambda = 3$ variants due to their slow performance for large intervals. In the remaining experiments NEWRMQ is therefore parametrized with by $s = 1024$ and $\Lambda = 2$. Figure 5 shows how much time is spent on the basic operations. Most time is spent on SELECT$_1$ and ACCESS of E and I. These operations consist mainly of memory accesses and get therefore more expensive for larger inputs due to address translation. Note that SELECT$_1$ is cheaper for smaller ranges due to caching effects and also the time spent in SCAN is notable for small query ranges. For larger query ranges SCAN is not triggered, as we can exclude the results from the fringe bocks by the optimizations presented in Algorithm 3.

Next, we compare NEWRMQ to the other implementations on RANDOM. The results in Figure 6 for the three competitors are consistent with the outcome of Navarro & Ferrada's study [5]. The experiment shows that the optimization for large query ranges, which avoids scanning the fringe blocks, is much more effective for NEWRMQ than for F&N'16, where it is only applied to the range min-max tree. We found that the range min-max tree is not necessarily the cause for many cache misses but for many cache references; see Figure 7 for detailed numbers. Applying SPARSETABLE and the tailored SELECT in NEWRMQ reduced the number of cache references significantly.

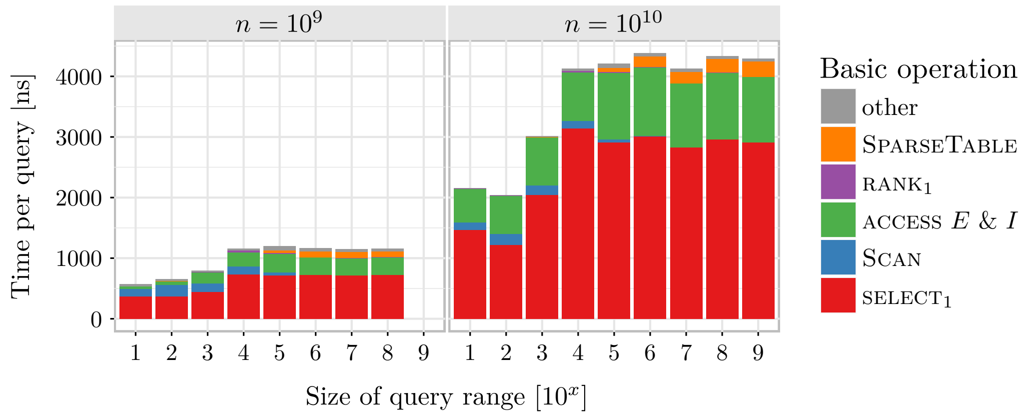


Figure 5 Query time breakdown for NEWRMQ obtained by measuring time spent in each basic operation.

Table 1 Statistics for LCP arrays of the full *Pizza&Chilli* texts.

	dblp	dna	english	sources
Depth of $\mathcal{C}_L^L(A)$	543	371	664	**765**
Depth of $\mathcal{C}_R^L(\overleftarrow{A})$	**54**	**120**	**132**	3232
Depth of suffix tree	124	305	148	3238
Ratio of avoided second SELECT calls (by optimization in Line 7 of Algo. 2)	88.32%	91.41%	87.31%	88.33%

Next, we explore the performance on pseudo-increasing and -decreasing inputs (INC-δ and DEC-δ). Figure 8 and Figure 9 show the results. As expected the performance of NEWRMQ is very similar on both inputs, since we minimize the height of the tree in both cases. For these inputs, it is very likely that the result of a query is located in the left or right fringe block. Therefore the optimization that avoid the scan of fringe blocks for large query ranges is not as effective as in the previous experiment.

Finally, we also consider the performance on a real-world application, namely the traversal of a suffix tree. Here we build our structure over the LCP array and use RMQs to implement the child operation for a node. A stack is used to maintain the ancestors of the currently traversed node. Query ranges – in non-degenerate suffix trees – are typically small and therefore the optimization in Line 7 in Algorithm 2 takes effect. Figure 10 depicts the timing results while Table 1 quantifies the saved operations and also reports the heights of the two Cartesian tree variants.

5 Conclusion

In this work we concerned ourselves with a practical solution for the range minimum query problem. In order to develop a fast solution that is also space efficient, we build upon previous theory and implementation ideas. We propose a new implementation that incorporates novel optimizations that improve the practical performance even further. Compared to existing solutions we replace the range min-max tree with a simpler recursive approach terminated by a sparse table.

Our experimental results show that our new implementation is up to three times faster than previous implementations, while retaining low space usage only slightly above the

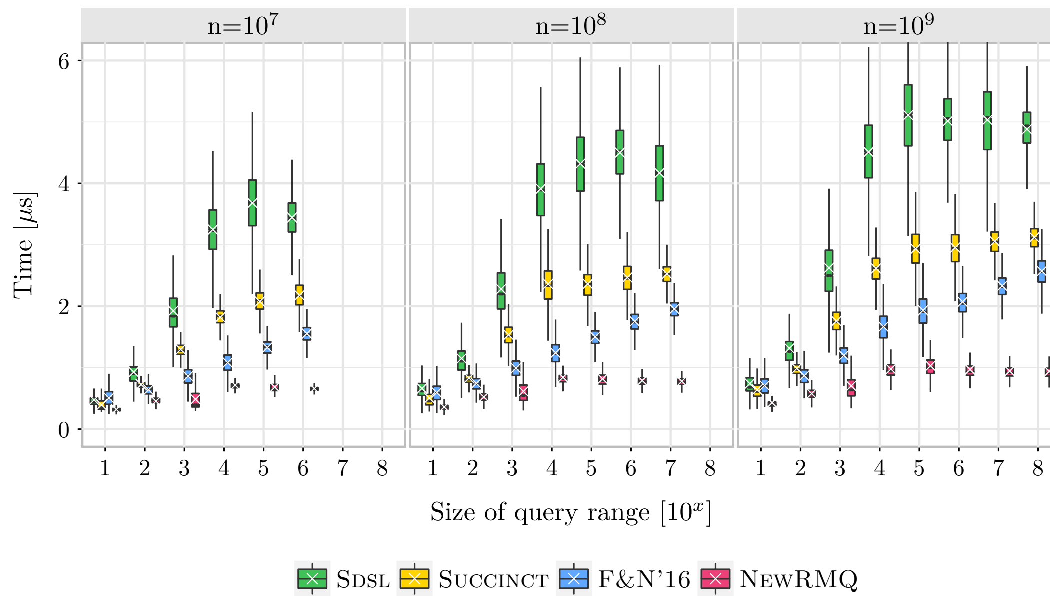


Figure 6 Query time distribution for all implementations on input RANDOM.

theoretical lower bound of 2 bits per input element. For all tested inputs, including both artificial uniform and pseudo-increasing/decreasing random sequences as well as a selected real-world application, we consistently outperform previous implementations.

References

1. D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. ALENEX*, pages 84–97. SIAM, 2010.
2. M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. LATIN*, pages 88–94. Springer, 2000.
3. O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.
4. Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. Succinct representations of binary trees for range minimum queries. In *Proc. CACOON*, pages 396–407, 2012.
5. H. Ferrada and G. Navarro. Improved range minimum queries. *J. Discrete Alg.*, 2016. To appear.
6. H. Ferrada and G. Navarro. Improved range minimum queries. In *Proc. DCC*, pages 516–525, 2016.
7. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *J. Exp. Algorithmics*, 13:12:1.12–12:1.31, February 2009. `doi:10.1145/1412228.1455268`.
8. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.
9. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
10. H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Proc. STOC*, pages 135–143. ACM, 1984.
11. S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337, 2014.

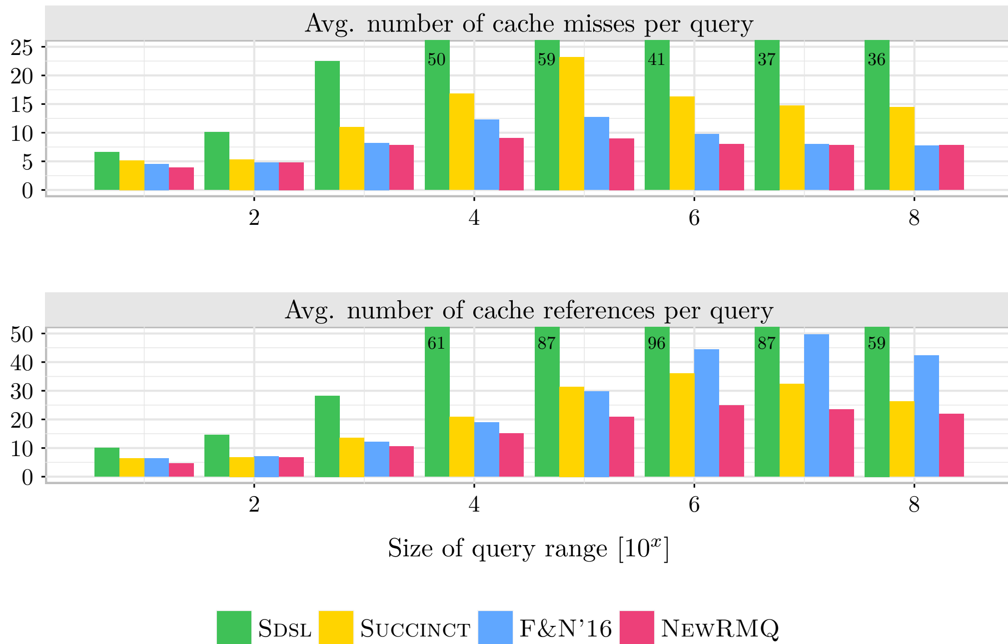


Figure 7 Cache access statistics for all implementations on input RANDOM of size $n = 10^9$.

12 R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Proc. SEA*, pages 5–17, 2013.

13 B. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *Proc. WWW*, pages 583–594. ACM, 2013.

14 G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.

15 J. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

16 G. Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

17 G. Navarro and K. Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16:1–16:39, May 2014. doi:10.1145/2601073.

18 K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

19 J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

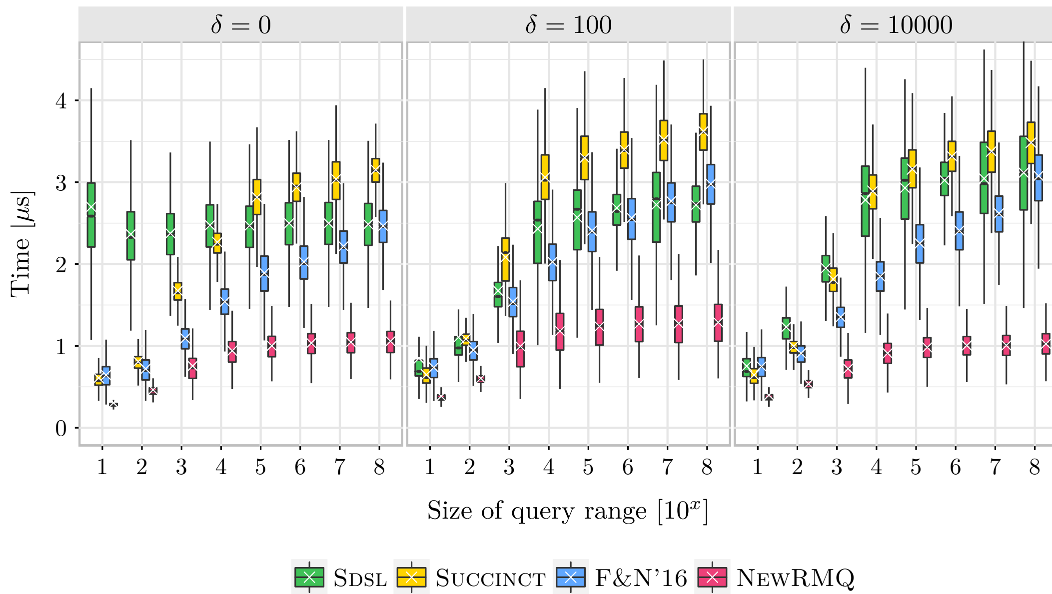


Figure 8 Query time distribution for pseudo-increasing input arrays of size $n = 10^9$.

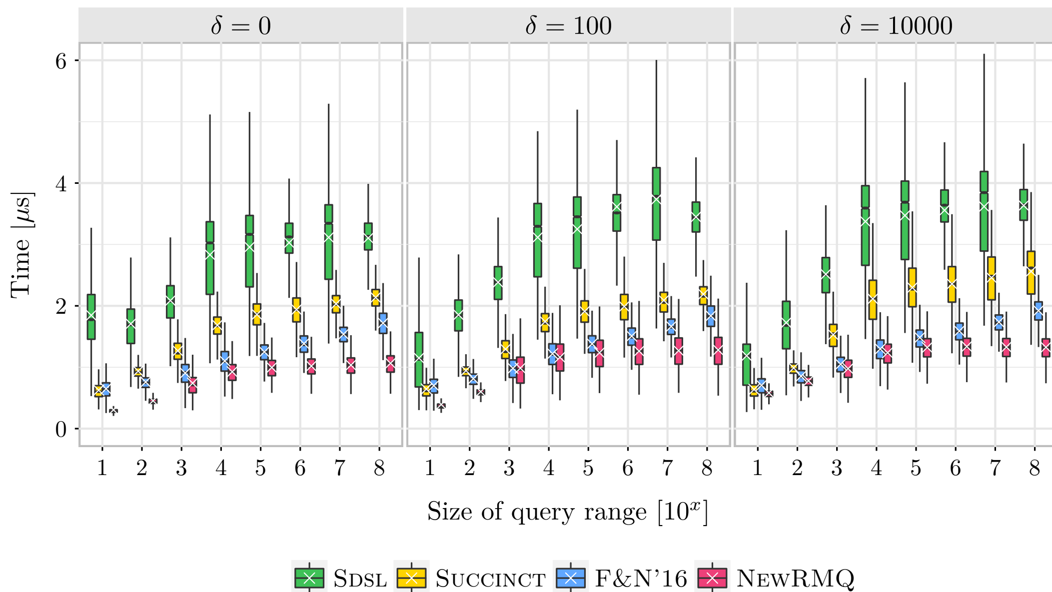


Figure 9 Query time distribution for pseudo-decreasing input arrays of size $n = 10^9$.

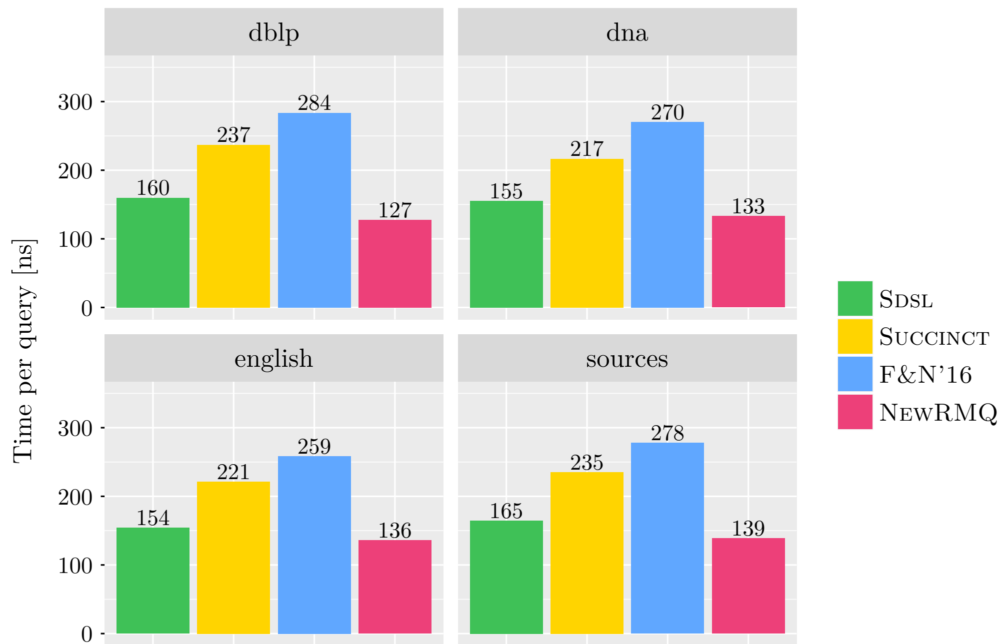


Figure 10 Application benchmark: DFS traversal of a suffix tree. RMQs over the LCP-array are used to calculate the children of a node. We used different texts of the *Pizza&Chilli* corpus[7].

Table 2 Memory consumption dependent on experiment, input size, and implementation. For Figure 5 and Figure 8 we show the data for strictly increasing respectively decreasing sequences.

Implementation	Space in bits per element with varying n					
	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$	$n = 10^8$	$n = 10^9$
Data of Fig. 4						
$s = 1024,\ \Lambda = 1$	2.41	2.33	2.37	2.44	2.54	2.65
$s = 1024,\ \Lambda = 2$	2.41	2.18	2.16	2.16	2.16	2.17
$s = 1024,\ \Lambda = 3$	2.41	2.18	2.16	2.16	2.16	2.16
$s = 2048,\ \Lambda = 1$	2.32	2.23	2.24	2.27	2.32	2.37
$s = 2048,\ \Lambda = 2$	2.32	2.16	2.14	2.14	2.14	2.15
$s = 2048,\ \Lambda = 3$	2.32	2.16	2.14	2.14	2.14	2.14
$s = 4096,\ \Lambda = 1$	2.27	2.18	2.18	2.19	2.21	2.24
$s = 4096,\ \Lambda = 2$	2.27	2.18	2.14	2.14	2.14	2.14
$s = 4096,\ \Lambda = 3$	2.27	2.18	2.14	2.13	2.13	2.13
Data of Fig. 6						
SDSL	2.64	3.26	2.61	2.55	2.54	2.54
SUCCINCT	2.80	2.71	2.70	2.71	2.71	2.70
F&N'16	4.52	2.31	2.10	2.09	2.10	2.10
NEWRMQ	2.41	2.18	2.16	2.16	2.16	2.17
Data of Fig. 8						
SDSL	2.62	3.26	2.60	2.54	2.53	2.53
SUCCINCT	2.80	2.71	2.70	2.71	2.71	2.70
F&N'16	4.63	2.43	2.24	2.26	2.28	2.31
NEWRMQ	2.41	2.17	2.16	2.15	2.15	2.16
Data of Fig. 9						
SDSL	2.64	3.27	2.62	2.55	2.54	2.54
SUCCINCT	2.80	2.71	2.70	2.71	2.71	2.70
F&N'16	4.51	2.29	2.07	2.05	2.06	2.06
NEWRMQ	2.41	2.17	2.16	2.15	2.15	2.16

Compression with the tudocomp Framework

Patrick Dinklage[1], Johannes Fischer[2], Dominik Köppl[3], Marvin Löbel[4], and Kunihiko Sadakane[5]

1 **Department of Computer Science, TU Dortmund, Dortmund, Germany**
pdinklag@gmail.com

2 **Department of Computer Science, TU Dortmund, Dortmund, Germany**
johannes.fischer@cs.tu-dortmund.de

3 **Department of Computer Science, TU Dortmund, Dortmund, Germany**
dominik.koeppl@tu-dortmund.de

4 **Department of Computer Science, TU Dortmund, Dortmund, Germany**
loebel.marvin@gmail.com

5 **Grad. School of Inf. Science and Technology, University of Tokyo, Tokyo, Japan**
sada@mist.i.u-tokyo.ac.jp

Abstract

We present a framework facilitating the implementation and comparison of text compression algorithms. We evaluate its features by a case study on two novel compression algorithms based on the Lempel-Ziv compression schemes that perform well on highly repetitive texts.

1998 ACM Subject Classification D.3.3 Frameworks, D.2.2 Software Libraries

Keywords and phrases lossless compression, compression framework, compression library, algorithm engineering, application of string algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.13

1 Introduction

Engineering novel compression algorithms is a relevant topic, shown by recent approaches like bc-zip [7], Brotli [1], or Zstandard[1]. Engineers of data compression algorithms face the fact that it is cumbersome (a) to build a new compression program from scratch, and (b) to evaluate and benchmark a compression algorithm against other algorithms objectively. We present the highly modular compression framework tudocomp that addresses both problems. To tackle problem (a), tudocomp contains standard techniques like VByte [28], Elias-γ/δ, or Huffman coding. To tackle problem (b), it provides automatic testing and benchmarking against external programs and implemented standard compressors like Lempel-Ziv compressors. As a case study, we present the two novel compression algorithms lcpcomp and LZ78U, their implementations in tudocomp, and their evaluations with tudocomp. lcpcomp is based on Lempel-Ziv 77, substituting greedily the longest remaining repeated substring. LZ78U is based on Lempel-Ziv 78, with the main difference that it allows a factor to introduce multiple new characters.

[1] https://github.com/facebook/zstd

© Patrick Dinklage, Johannes Fischer, Dominik Köppl, Marvin Löbel, and Kunihiko Sadakane;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 13; pp. 13:1–13:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Related Work

There are many[2] compression benchmark websites measuring compression programs on a given test corpus. Although the compression ratio of a novel compression program can be compared with the ratios of the programs listed on these websites, we cannot infer which program runs faster or more memory efficiently if these programs have not been compiled and run on the same machine. Efforts in facilitating this kind of comparison have been made by wrapping the source code of different compression algorithms in a single executable that benchmarks the algorithms on the same machine with the same compile flags. Examples include lzbench[3] and Squash[4].

Considering frameworks aiming at easing the comparison *and* implementation of new compression algorithms, we are only aware of the C++98 library ExCom [14]. The library contains a collection of compression algorithms. These algorithms can be used as components for a compression pipeline. However, ExCom does not provide the same flexibility as we had in mind; it provides only character-wise pipelines, i.e., it does no bitwise transmission of data. Its design does not use meta-programming features; a header-only library has more potential for optimization since the compiler can inline header-implemented (possibly performance critical) functions easily.

A broader focus is set in Giuseppe Ottaviano's succinct library [13] and Simon Gog's Succinct Data Structure Library 2.0 (SDSL) [12]. These two libraries provide integer coders and helper functions for working on the bit level.

1.2 Our Results/Approach

Our lossless compression framework ***tudocomp*** aims at supporting and facilitating the implementation of novel compression algorithms. The philosophy behind tudocomp is to support building a pipeline of modules that transforms an input to a compressed binary output. This pipeline has to be flexible: appending, exchanging and removing a module in the pipeline in a plug-and-play manner is in the main focus of the design of tudocomp. Even a module itself can be refined into submodules.

To this end, tudocomp is written in modern C++14. On the one hand, the language allows us to write compile time optimized code due to its meta programming paradigm. On the other hand, its fine-grained memory management mechanisms support controlling and monitoring the memory footprint in detail. We provide a tutorial, an exhaustive documentation of the API, and the source code at `http://tudocomp.org` with the permissive Apache License 2.0 to encourage developers to use and foster the framework.

In order to demonstrate its usefulness, we added reference implementations of common compression and encoding schemes (see Section 2). On top of that, we present two novel algorithms (see Section 3) which we have implemented in our framework. We give a detailed evaluation of these algorithms in Section 4, thereby exposing the benchmarking and the visualization tools of tudocomp.

---

[2] E.g., `http://www.squeezechart.com` or `http://www.maximumcompression.com`.
[3] `https://github.com/inikep/lzbench`
[4] `https://quixdb.github.io/squash-benchmark`

2 Description of the tudocomp Framework

On the topmost abstraction level, tudocomp defines the abstract types `Compressor` and `Coder`. A ***compressor*** transforms an input into an output so that the input can be losslessly restored from the output by the corresponding ***decompressor***. A ***coder*** takes an elementary data type like a character and writes it to a compressed bit sequence. As with compressors, each coder is accompanied by a ***decoder*** taking care of restoring the original data from its compressed bit sequence. By design, a coder can take the role of a compressor, but a compressor may not be suitable as a coder (e.g., a compressor that needs random access on the whole input).

tudocomp provides implementations of the compressors and the coders shown in the tables below. Each compressor and coder gets an identifier (right column of each table).

Compressors	
BWT	`bwt`
Coder wrapper	`encode`
LCPComp (Section 3.2)	`lcpcomp`
LZ77 (Def. 1), LZSS [25] output	`lzss_lcp`
LZ78 (Def. 2)	`lz78`
LZ78U (Section 3.3)	`lz78u`
LZW [27]	`lzw`
Move-To-Front	`mtf`
Re-Pair [20]	`repair`
Run-Length-Encoding	`rle`

Integer Coders	
Bit-Compact Coder	`bit`
Elias-γ [6]	`gamma`
Elias-δ [6]	`delta`

String Coders	
Canonical Huffman Coder [29]	`huff`
A Custom Static Low Entropy Encoder (Section 3.2)	`sle`

The behavior of a compressor or coder can be modified by passing different parameters. A parameter can be an elementary data type like an integer, but it can also be an instance of a class that specifies certain subtasks like integer coding. For instance, the compressor `lzss_lcp(threshold, coder)` takes an integer `threshold` and a `coder` (to code an LZ77 factor) as parameters. The coder is supplied as a parameter such that the compressor can call the coder directly (instead of alternatively piping the output of `lzss_lcp` to a coder).

The support of class parameters eases the deployment of the design pattern ***strategy*** [11]. A strategy determines what algorithm or data structure is used to achieve a compressor-specific task.

Library and Command Line Tool. tudocomp consists of two major components: a standalone compression library and a command line tool `tdc`. The library contains the core interfaces and implementations of the aforementioned compressors and coders. The tool `tdc` exposes the library's functionality in form of an executable that can run compressors directly on the command line. It allows the user to select a compressor by its identifier and to pass parameters to it, i.e., the user can specify the exact compression strategy *at runtime*.

Example. For instance, the LZ78U compressor (Section 3.3) expects a compression strategy, an integer coder, and an integer variable specifying a threshold. Its strategy can define parameters by itself, like which string coder to use. A valid call is `./tdc -a 'lz78u(coder = bit, comp = buffering(string_coder = huff), threshold = 3)' input.txt -o output.tdc`, where `tdc` compresses the file `input.txt` and stores the compressed bit sequence in the file `output.tdc`. To this end, it uses the compressor `lz78u` parametrized by

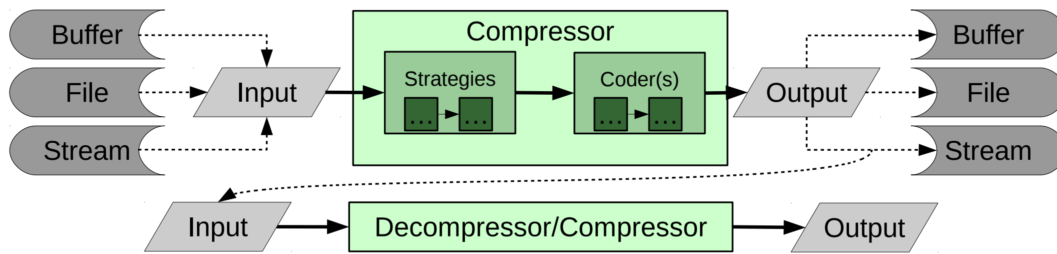


Figure 1 Flowchart of a possible compression pipeline. The compressors of tudocomp work with abstract data types for input and output, i.e., a compressor is unaware of whether the input or the output is a file, is stored in memory, or is accessed using a stream. A compressor can follow one or more compression strategies that can have (nested) parameters. Usually, a compressor is parametrized with one or more coders (e.g., for different integer ranges or strings) that produce the final output.

the coder `bit` for integer values, by the compression strategy `buffering` with `huff` to code strings, and by a threshold value of 3. Note that `coder` and `string_coder` are parameters for two independently selectable coders. When selecting a coder we have to pay attention that a static entropy coder like `huff` needs to parse its input in advance (to generate a codeword for each occurring character). To this end, we can only apply the coder `huff` with a compression strategy that buffers the output (for `lz78u` this strategy is called `buffering`). To stream the output (i.e., the opposite of buffering the complete output in RAM), we can use the alternative strategy `streaming`. This strategy also requires a coder, but contrary to the buffering strategy, that coder does not need to look at the complete output (e.g., universal codes like `gamma`).

In this fashion, we can build more sophisticated compression pipelines like `lzma` applying different coders for literals, pointers, and lengths. Each coder is unaware of the other coders, as if every coder was processing an independent stream.

Decompression. After compressing an input using a certain compression strategy, the tool adds a header to the compressed file so that it can decompress it without the need for specifying the compression strategy again. However, this behavior can be overruled by explicitly specifying a decompression strategy, e.g., in order to test different decompression strategies.

Helper classes. tudocomp provides several classes for easing common tasks when engineering a new compression algorithm, like the computation of SA, ISA or LCP. tudocomp generates SA with `divsufsort`[5], and LCP with the Φ-algorithm [17]. The arrays SA, ISA, and LCP can be stored in plain arrays or in packed arrays with a bit width of $\lceil \lg n \rceil$ (where n is the length of the input text), i.e., in a ***bit-compact representation***. We provide the modes `plain`, `compressed`, and `delayed` to describe when/whether a data structure should be stored in a bit-compact representation: In `plain` mode, all data structures are stored in plain arrays; in `compressed` mode, all data structures are built in a bit-compact representation. In `delayed` mode, tudocomp first builds a data structure A in a plain array; when all other data structures are built whose constructions depended on A, A gets transformed into a bit-compact representation. While `direct` and `compressed` are the fastest or the

[5] `https://github.com/y-256/libdivsufsort`

memory-friendliest modes, respectively, the data structures produced by `delayed` are the same as `compressed`, though `delayed` is faster than `compressed`.

If more elaborated algorithms are desired (e.g., for producing compressed data structures like the compressed suffix array), it is easy to use tudocomp in conjunction with SDSL for which we provide an easy binding.

Combining streaming and offline approaches. A compressor can stream its input (online approach) or request the input to be loaded into memory (offline approach). Compressors can be chained to build a pipeline of multiple compression modules, like as in Figure 1.

2.1 Example Implementation of a Compressor

(a) C++ Source Code

```
1  #include <tudocomp/tudocomp.hpp>
2  class BWTComp : public Compressor {
3    public: static Meta meta() {
4      Meta m("compressor", "bwt");
5      m.option("ds").templated<TextDS<>>();
6      m.needs_sentinel_terminator();
7      return m; }
8    using Compressor::Compressor;
9    void compress(Input& in, Output& out) {
10     auto o = out.as_stream();
11     auto i = in.as_view();
12     TextDS<> t(env().env_for_option("ds"),i);
13     const auto& sa = t.require_sa();
14     for(size_t j = 0; j < t.size(); ++j)
15       o << ((sa[j] != 0) ? t[sa[j] - 1]
16                           : t[t.size() - 1]);
17   }
18   void decompress(Input&, Output&){/*[...]*/}
19 };
```

(b) Execution with `tdc`

```
1  > echo -n 'aaababaaabaababa' > ex.txt
2  > ./tdc -a bwt -o bwt.tdc ex.txt
3  > hexdump -v -e '"%-2_c"' bwt.tdc
4  b w t % a b b \0a b a b b a a a a a a a a
5  > ./tdc -a 'bwt:rle' -o rle.tdc ex.txt
6  > hexdump -v -e '"%-3_c"' rle.tdc
7  b  w  t  :  r  l  e  %  a  b  b  \0 \0 a  b
     a  b  b  \0 a  a  006
```

The source code (a) on the left implements a compressor that computes the Burrows-Wheeler transform (BWT) (see Section 3.1) of an input. To this end, it loads the input into memory using (line 11) `in.as_view()` and computes the suffix array using (line 13) `t.require_sa()`. In the function `meta`, we state that we assume the unique terminal symbol (represented by the byte '`\0`') as part of the text, and that we want to register the class `BWTComp` as a `Compressor` with the identifier `bwt`. By doing so, we can call the compressor directly in the command line tool `tdc` using the argument `-a bwt`. In the shell code (b) on the left, you can see how we produced the BWT of our running example. The program `hexdump` outputs each character of a file such that non-visible characters are escaped. A `%`-sign separates the header from the body in the output. Next, we use the binary composition operator `:` connecting the output of its left operand with the input of its right operand. In the shell code, this operator pipes the output of `bwt` to the run-length encoding compressor `rle`, which transforms a substring $\mathtt{aa}\underbrace{\mathtt{a}\cdots\mathtt{a}}_{m\text{ times}}$ to $\mathtt{aa}m$ with $m \geq 0$ encoded in VByte (the output is a *byte* sequence).

(c) Assembling a compression pipeline

```
1  > ./tdc -a bwt -o bwt.tdc pc_english.200MB
2  > ./tdc -a 'bwt:rle:mtf:encode(huff)' -o bzip
     .tdc pc_english.200MB
3  > stat -c"%s␣%n" pc_english.200MB *.tdc
4  209715200 pc_english.200MB
5  209715209 bwt.tdc
6  66912437 bzip.tdc
```

Finally, the compressor `bwt` can be used as part of a pipeline to achieve good compression quality: Given a move-to-front compressor `mtf` and a Huffman coder `huff`, we can build a chain `bwt:rle:mtf:encode(huff)`. The compressor `encode` is a wrapper that turns a coder into a compressor. The last code fragment (c) on the left shows the calls of this pipeline and a call of `bwt` only. Using `stat`, we measure the file sizes (in bytes) of the input PC-ENGLISH (see Section 4) and both outputs.

2.2 Specific Features

tudocomp excels with the following additional properties:

Few Build Requirements. To deploy tudocomp, the build management software `cmake`, the version control system `git`, Python 3, and a C++14 compiler are required. `cmake` automatically downloads and builds other third-party software components like the SDSL. We tested the build process on Unix-like build environments, namely Debian Jessie, Ubuntu Xenial, Arch Linux 2016, and the Ubuntu shell on Windows 10.

Unit Tests. tudocomp offers semi-automatic unit tests. For a registered compressor, tudocomp can automatically generate test cases that check whether the compressor can compress and decompress a set of selected inputs successfully. These inputs include border cases like the empty string, a run of the same character, samples on various subranges in UTF-8, Fibonacci strings, Thue-Morse strings, and strings with a high number of runs [22]. These strings can be generated on-the-fly by `tdc` as an alternative input.

Type Inferences. The C++ standard does neither provide a syntax for constraining type parameters (like generic type bounding in Java) nor for querying properties of a class at runtime (i.e., reflection). To address this syntactic lack, we augment each class exposed to `tdc` and to the unit tests with a so-called ***type***. A type is a string identifier. We expect that classes with the same type provide the same public methods. Types resemble *interfaces* of Java, but contrary to those, they are not subject to polymorphism. Common types in our framework are `Compressor` and `Coder`. The idea is that, given a compressor that accepts a `Coder` as a parameter, it should accept all classes of type `Coder`. To this end, each typed class is augmented with an identifier and a description of all parameters that the class accepts. All typed classes are exposed by the tool `tdc` that calls a typed class by its identifier with the described parameters. Types provide a uniform, but simple declaration of all parameters (e.g., integer values, or strategy classes). The aforementioned exemplaric call of `lz78u` at the beginning of Section 2 illustrates the uniform declaration of the parameters of a compressor.

Evaluation tools. To evaluate a compressor pipeline, tudocomp provides several tools to facilitate measuring the compression ratio, the running time, and the memory consumption. By adding `--stats` to the parameters of `tdc`, the tool monitors these measurement parameters: It additionally tracks the running time and the memory consumption of the data structures in all phases. A phase is a self-defined code division like a pre-processing phase, or an encoding phase. Each phase can collect self-defined statistics like the number of generated factors. All measured data is collected in a JSON file that can be visualized by the web application found at `http://tudocomp.org/charter`. An example is given in Figure 6.

In addition, we have a command line ***comparison tool*** called `compare.py` that runs a predefined set of compression programs (that can be tudocomp compressors or external compression programs). Its primary usage is to compare tudocomp compression algorithms with external compression programs. It monitors the memory usage with the tool `valgrind -tool=massif -pages-as-heap=yes`. This tool is significantly slower than running `tdc` with `--stats`.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	a	a	a	b	a	b	a	a	a	b	a	a	b	a	b	a	$
SA[i]	17	16	7	1	8	11	2	14	5	9	12	3	15	6	10	13	4
ISA[i]	4	7	12	17	9	14	3	5	10	15	6	11	16	8	13	2	1
LCP[i]	-	0	1	5	2	4	6	1	3	4	3	5	0	2	3	2	4
BWT[i]	a	b	b	$	a	b	a	b	b	a	a	a	a	a	a	a	a

Figure 2 Suffix array, inverse suffix array, LCP array and BWT of the running example.

3 New Compression Algorithms

With the aid of tudocomp, it is easy to implement new compression algorithms. We demonstrate this by introducing two novel compression algorithms: ***lcpcomp*** and ***LZ78U***. To this end, we first recall some definitions.

3.1 Theoretical Background

Let Σ denote an integer alphabet of size $\sigma = |\Sigma| \leq n^{\mathcal{O}(1)}$ for a natural number n. We call an element $T \in \Sigma^*$ a ***string***. The empty string is ϵ with $|\epsilon| = 0$. Given $x, y, z \in \Sigma^*$ with $T = xyz$, then x, y and z are called a ***prefix***, ***substring*** and ***suffix*** of T, respectively. We call $T[i..]$ the i-th suffix of T, and denote a substring $T[i] \cdots T[j]$ with $T[i..j]$.

For the rest of the article, we take a string T of length n. We assume that $T[n]$ is a special character $\$ \notin \Sigma$ smaller than all characters of Σ so that no suffix of T is a prefix of another suffix of T.

SA and ISA denote the suffix array [21] and the inverse suffix array of T, respectively. LCP$[2..n]$ is an array such that LCP$[i]$ is the length of the longest common prefix of the *lexicographically* i-th smallest suffix with its lexicographic predecessor for $i = 2, \ldots, n$. The BWT [3] of T is the string BWT with

$$\mathsf{BWT}[j] = \begin{cases} T[n] & \text{if } \mathsf{SA}[j] = 1, \\ T[\mathsf{SA}[j] - 1] & \text{otherwise,} \end{cases}$$

for $1 \leq j \leq n$. The arrays SA, ISA, LCP and BWT can be constructed in time linear to the number of characters of T [18].

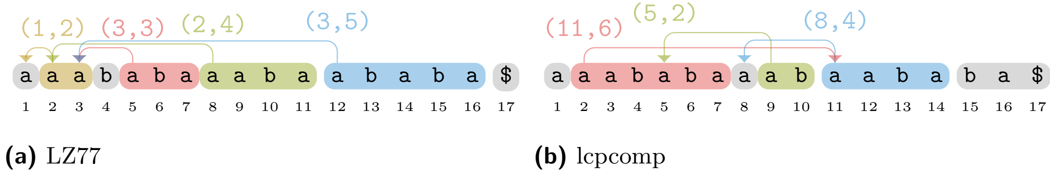
As a running example, we take the text $T :=$ aaababaaabaababa$. The arrays SA, LCP and BWT of this example text are shown in Figure 2.

Given a bit vector B with length $|B|$, the operation $B.\mathrm{rank}_1(i)$ counts the number of '1'-bits in $B[1..i]$, and the operation $B.\mathrm{select}_1(i)$ yields the position of the i-th '1' in B.

There are data structures [15, 4] that can answer rank and select queries on B in constant time, respectively. Each of them uses $o(|B|)$ additional bits of space, and both can be built in $\mathcal{O}(|B|)$ time.

The ***suffix trie*** of T is the trie of all suffixes of T. The ***suffix tree*** [26] of T, denoted by ST, is the tree obtained by compacting the suffix trie of T. It has n leaves and at most n internal nodes. The string stored in an edge e is called the ***edge label*** of e, and denoted by $\lambda(e)$. The ***string depth*** of a node v is the length of the concatenation of all edge labels on the path from the root to v. The leaf corresponding to the i-th suffix is labeled with i.

Each node of the suffix tree is uniquely identified by its pre-order number. We can store the suffix tree topology in a bit vector (e.g., DFUDS [2] or BP [15, 23]) such that rank and select queries enable us to address a node by its pre-order number in constant time. If the



(a) LZ77 **(b)** lcpcomp

Figure 3 References of the (a) LZ77 factorization with the threshold $t = 2$, and of the (b) lcpcomp factorization with the same threshold. The output of the LZ77 and the lcpcomp algorithms are `a(1,2)b(3,3)(2,4)(3,5)$` and `a(11,6)a(5,2)(8,4)ba$`, respectively.

context is clear, we implicitly convert an ST node to its pre-order number, and vice versa. We will use the following constant time operations on the suffix tree:

- parent(v) selects the parent of the node v,
- level-anc(ℓ, d) selects the ancestor of the leaf ℓ at depth d (level ancestor query), and
- leaf-select(i) selects the i-th leaf (in lexicographic order).

A ***factorization*** of T of size z partitions T into z substrings $T = F_1 \cdots F_z$. These substrings are called ***factors***. In particular, we have:

▶ **Definition 1.** A factorization $F_1 \cdots F_z = T$ is called the ***Lempel-Ziv-77 (LZ77) factorization*** [30] of T with a threshold $t \geq 1$ iff F_x is either the longest substring of length at least t occurring at least twice in $F_1 \cdots F_x$, or, if such a substring does not exist, a single character. We merge successive occurrences of the latter type of factors to a single factor and call it a ***remaining substring***.

The usual definition of the LZ77 factorization fixes $t = 1$. We introduced the version with a threshold to make the comparison with lcpcomp (Section 3.2) fairer.

▶ **Definition 2.** A factorization $F_1 \cdots F_z = T$ is called the ***Lempel-Ziv-78 (LZ78) factorization*** [31] of T iff $F_x = F_y \cdot c$ with $F_y = \text{argmax}_{S \in \{F_y : y < x\} \cup \{\epsilon\}} |S|$ and $c \in \Sigma$ for all $1 \leq x \leq z$.

3.2 lcpcomp

The idea of lcpcomp is to search for long repeated substrings and substitute one of their occurrences with a reference to the other. Large values in the LCP-array indicate such long repeated substrings. There are two major differences to the LZ77 compression scheme: (1) while LZ77 only allows back-references, lcpcomp allows both back *and forward* references; and (2) LZ77 factorizes T greedily from left to right, whereas lcpcomp makes substitutions at *arbitrary* positions in the text, greedily chosen such that the number of substituted characters is maximized. This process is repeated until all remaining repeated substrings are shorter than a threshold t. On termination, lcpcomp has generated a factorization $T = F_1 \cdots F_z$, where each F_j is either a remaining substring, or a reference (i, ℓ) with the intended meaning “copy ℓ characters from position i” (see Figure 3b for an example).

Algorithm. The LCP array stores the longest common prefix of two lexicographically neighboring suffixes. The largest entries in the LCP array correspond to the longest substrings of the text that have at least two occurrences. Given a suffix $T[\mathsf{SA}[i]\,..]$ whose entry $\mathsf{LCP}[i]$ is maximal among all other values in LCP, we know that $T[\mathsf{SA}[i]\,..\,\mathsf{SA}[i] + \mathsf{LCP}[i] - 1] = T[\mathsf{SA}[i-1]\,..\,\mathsf{SA}[i-1] + \mathsf{LCP}[i] - 1]$, i.e., we can substitute $T[\mathsf{SA}[i]\,..\,\mathsf{SA}[i] + \mathsf{LCP}[i] - 1]$ with the reference $(\mathsf{SA}[i-1]\,, \mathsf{LCP}[i])$. In order to find a suffix whose LCP entry is maximal, we need a data structure that maintains suffixes ordered by their corresponding LCP values. We

use a maximum heap for this task. To this end, the heap stores suffix array indices whose keys are their LCP values (i.e., insert i with key $\mathsf{LCP}[i]$, $2 \leq i \leq n$). The heap stores only those indices whose keys are at least t.

While the heap is not empty, we do the following:

1. Remove the maximum from the heap; let i be its value.
2. Report the reference $(\mathsf{SA}[i-1], \mathsf{LCP}[i])$ and the position $\mathsf{SA}[i]$ as a triplet $(\mathsf{SA}[i-1], \mathsf{LCP}[i], \mathsf{SA}[i])$.
3. For every $1 \leq k \leq \mathsf{LCP}[i]-1$, remove the entry $\mathsf{ISA}[\mathsf{SA}[i]+k]$ from the heap (as these positions are covered by the reported reference).
4. Decrease the keys of all entries j with $\mathsf{SA}[i]-\mathsf{LCP}[i] \leq \mathsf{SA}[j] < \mathsf{SA}[i]$ to $\min(\mathsf{LCP}[j], \mathsf{SA}[i]-\mathsf{SA}[j])$. (If a key becomes smaller than t, remove the element from the heap.) By doing so, we prevent the substitution of a substring of $T[\mathsf{SA}[i]..\mathsf{SA}[i]+\mathsf{LCP}[i]-1]$ at a later time.

```
1  template<class text_t>
2  class MaxHeapStrategy : public Algorithm {
3   public: static Meta meta() {
4    Meta m("lcpcomp_strategy", "heap");
5    return m; }
6   using Algorithm::Algorithm;
7   void create_factor(size_t pos, size_t src,
        size_t len);
8   void factorize(text_t& text, size_t t) {
9    text.require(text_t::SA | text_t::ISA |
        text_t::LCP);
10   auto& sa = text.require_sa();
11   auto& isa = text.require_isa();
12   auto lcpp = text.release_lcp()->relinquish
        ();
13   auto& lcp = *lcpp;
14   ArrayMaxHeap<typename text_t::lcp_type::
        data_type> heap(lcp, lcp.size(), lcp.
        size());
15   for(size_t i = 1; i < lcp.size(); ++i)
16    if(lcp[i] >= t) heap.insert(i);
17   while(heap.size() > 0) {
18    size_t i = heap.top(), fpos = sa[i],
19        fsrc = sa[i-1], flen = heap.key(i);
20    create_factor(fpos, fsrc, flen);
21    for(size_t k=0; k < flen; k++)
22     heap.remove(isa[fpos + k]);
23    for(size_t k=0;k < flen && fpos > k;k++) {
24     size_t s = fpos - k - 1;
25     size_t j = isa[s];
26     if(heap.contains(j)) {
27      if(s + lcp[j] > fpos) {
28       size_t l = fpos - s;
29       if(l >= t)
30        heap.decrease_key(j, l);
31       else heap.remove(j);
32 }}}}}};
```

As an invariant, the key ℓ of a suffix array index i stored in the heap will always be the maximal number of characters such that $T[i..i+\ell-1]$ occurs at least twice in the *remaining* text.

The reported triplets are collected in a list. To compute the final output, we sort the triplets by their third component (storing the starting position of the substring substituted by the reference stored in the first two components). We then scan simultaneously over the list and the text to generate the output. Figure 4 demonstrates how the lcpcomp factorization of the running example is done step-by-step.

The code on the left implements the compression strategy of lcpcomp that uses a maximum heap. We transfered the code from the compressor class to a strategy class since the lcpcomp compression scheme can be implemented in different ways. Each strategy receives a text. Its goal is to compute all factors (created by the `create_factor` method). In the depicted strategy, we use a maximum heap to find all factors. The heap is implemented in the class `ArrayMaxHeap`. An instance of that class stores an array A of keys and an array heap maintaining (key-value)-pairs of the form $(A[i], i)$ with the order $(A[i], i) < (A[j], j) :\Leftrightarrow A[i] < A[j]$. To access a specific element in the heap by its value, the class has an additional array storing the position of each value in the heap.

Although a reference r can refer to a substring that has been substituted by another reference after the creation of r, in Lem. 4 (Appendix), we show that it is always possible to restore the text.

Time Analysis. We insert at most n values into the heap. No value is inserted again. Finally, we use the following lemma to get a running time of $\mathcal{O}(n \lg n)$:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	a	a	a	b	a	b	a	a	a	b	a	a	b	a	b	a	$
SA$[i]$	17	16	7	1	8	11	2	14	5	9	12	3	15	6	10	13	4
LCP$[i]$	–	0	1	5	2	4	6	1	3	4	3	5	0	2	3	2	4
LCP$^1[i]$	–	0	0	1	2	4	0	1	0	4	3	0	0	0	3	2	0
LCP$^2[i]$	–	0	0	1	2	0	0	0	0	2	0	0	0	0	1	0	0
LCP$^3[i]$	–	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4 Step-by-step computation of the lcpcomp compression scheme in Figure 3b. We scan for the largest LCP value in LCP and overwrite values in LCP instead of using a heap. Each row LCP$^i[i]$ shows the LCP array after computing a substitution. The LCP value of the starting position of the selected largest repeated substring has a green border. The updated values are colored, either due to deletion (red) or key reduction (blue). Ties are broken arbitrarily. The number of red zeros in each row is equal to the number above the green bordered zero in the corresponding row minus one.

▶ **Lemma 3.** *The key of a suffix array entry is decreased at most once.*

Proof. Let us denote the key of a value i stored in the heap by $K[i]$. Assume that we have decreased the key $K[j]$ of some value j stored in the heap after we have substituted a substring $T[i..i+\ell-1]$ with a reference. It holds that $K[j] = \mathsf{SA}[i] - \mathsf{SA}[j] - 1 > \mathsf{SA}[i] - \mathsf{SA}[j] - 1 - m \geq K[\mathsf{ISA}[\mathsf{SA}[j]+m]]$ for all m with $1 \leq m \leq K[j]$, i.e., there is no suffix array entry that can decrease the key of j again. ◀

3.2.1 Decompression

Decompressing lcpcomp-compressed data is harder than decompressing LZ77, since references in lcpcomp can refer to positions that have not yet been decoded. Figure 3 depicts the references built on our running example by arrows.

In order to cope with this problem, we add, for each position i of the original text, a list L_i storing the text positions waiting for this text position getting decompressed.

First, we determine the original text size (the compressor stores it as a VByte before the output of the factorization). Subsequently, while there is some compressed input, we do the following, using a counting variable i as a cursor in the text that we are going to rebuild:

- If the input is a character c, we write $T[i] \leftarrow c$, and increment i by one.
- If the input is a reference consisting of a position s and a length ℓ, we check whether $T[s+j]$ is already decoded, for each j with $0 \leq j \leq \ell - 1$:
  - If it is, then we can restore $T[i+j] \leftarrow T[s+j]$.
  - Otherwise, we add $i+j$ to the list L_{s+j}.

  In either case, we increment i by ℓ.

An additional procedure is needed to restore the text completely by processing the lists: On writing $T[i] \leftarrow c$ for some text position i and some character c, we further write $T[j] \leftarrow T[i]$ for each j stored in L_i (if L_j is not empty, we proceed recursively). Afterwards, we can delete L_i since it will be no longer needed. The decompression runs in $\mathcal{O}(n)$ time, since we perform a linear scan over the decompressed text, and each text position is visited at most twice.

3.2.2 Implementation Improvements

In this section, we present an $\mathcal{O}(n)$ time compression algorithm alternative to the heap strategy and a practical improvement of the decompression strategy.

Compression. This strategy computes an array A_ℓ storing all suffix array entries j with $\mathsf{LCP}[j] = \ell$, for each ℓ with $t \leq \ell \leq \max_k \mathsf{LCP}[k]$. To compute the references, we sequentially scan the arrays in decreasing order, starting with the array that stores the suffixes with the maximum LCP value. On substituting a substring $T[\mathsf{SA}[i] \mathinner{..} \mathsf{SA}[i] + \mathsf{LCP}[i] - 1]$ with the reference $(\mathsf{SA}[i-1], \mathsf{LCP}[i])$, we update the LCP array (instead of updating the keys in the heap). We set $\mathsf{LCP}[\mathsf{ISA}[\mathsf{SA}[i] + k]] \leftarrow 0$ for every $1 \leq k \leq \mathsf{LCP}[i] - 1$ (deletion), and $\mathsf{LCP}[j] \leftarrow \min(\mathsf{LCP}[j], \mathsf{SA}[i] - \mathsf{SA}[j])$ for every j with $\mathsf{ISA}[\mathsf{SA}[i] - \mathsf{LCP}[i]] \leq j < i$ (decrease key). Unlike the heap implementation, we do *not* delete an entry from the arrays. Instead, we look up the current LCP value of an element when we process it: Assume that we want to process $A_\ell[i]$. If $\mathsf{LCP}[A_\ell[i]] = \ell$, then we proceed as above. Otherwise, we have updated the LCP value of the suffix starting at position $A_\ell[i]$ to the value $\ell' := \mathsf{LCP}[A_\ell[i]] < \ell$. In this case, we append $A_\ell[i]$ to $A_{\ell'}$ (if $\ell' < t$, we do nothing), and skip computing the reference for $A_\ell[i]$. By doing so, we either omit the substring $A_\ell[i]$ if $\ell' < t$, or delay the processing of the value $A_\ell[i]$. A suffix array entry gets delayed at most once, analogously to Lemma 3. In total, the algorithm runs in $\mathcal{O}(n)$ time, since it performs basic arithmetic operations on each text position at most twice.

Decompression. We use a heuristic to improve the memory usage. The heuristic defers the creation of the lists L_i storing the text positions that are waiting for the position i to get decompressed. If a reference needs a substring that has not yet been decompressed, we store the reference in a list L. By doing so, we have reconstructed at least all substrings that have not been substituted by a reference during the compression. Subsequently, we try to decompress each reference stored in L, removing successfully decompressed references from L. If we repeat this step, more and more text positions can become restored. Clearly, after at most n iterations, we would have restored the original text completely, but this would cost us $\mathcal{O}(n^2)$ time. Instead, we run this algorithm only for a fixed number of times b. Afterwards, we mark all not yet decompressed positions in a bit vector B, and build a rank data structure on top of B. Next, we create a list L_i for each marked text position $B.\mathrm{rank}(i)$ as in the original algorithm. The difference to the original algorithm is that L_i now corresponds to $B.\mathrm{rank}(i)$. Finally, we run the original algorithm using the lists L_i to restore the remaining characters.

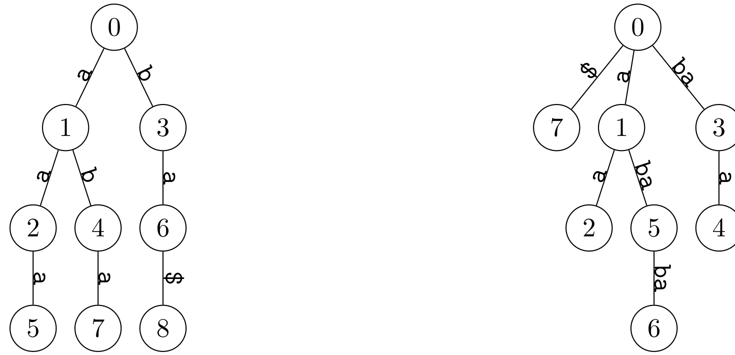
3.3 LZ78U

A factorization $F_1 \cdots F_z = T$ is called the ***LZ78U factorization*** of T iff $F_x := T[i..j+\ell]$ with $T[i..j] = \mathrm{argmax}_{S \in \{F_y : y < x\} \cup \{\epsilon\}} |S|$ and

$$\ell := \begin{cases} 1 \text{ if } T[i..j+1] \text{ is a unique substring of } T, \text{otherwise:} \\ 1 + \max\{\ell \in \mathbb{N}_0 \mid \forall k = 1, \ldots, \ell \nexists c \in \Sigma \setminus \{T[j+k+1]\} : T[i..j+k]c \text{ occurs in } T\}, \end{cases}$$

for all $1 \leq x \leq z$. Informally, we enlarge an LZ78 factor representing a repeated substring $T[i..i+\ell-1]$ to $T[i..i+\ell]$ as long as the number of occurrences of $T[i..i+\ell-1]$ and $T[i..i+\ell]$ are the same.

Having the LZ78U factorization $F_1, \ldots, F_z$ of T, we can output each factor F_x as a tuple (y, S_x) such that $F_x = F_y S_x$, where F_y $(0 \leq y < x)$ is the longest previous factor (set $F_0 := \epsilon$)



(a) LZ78-Trie **(b)** LZ78U-Tree

Figure 5 Dictionary trees of LZ78 and LZ78U. LZ78 factorizes our running example into `a|aa|b|ab|aaa|ba|aba|ba$`, where the vertical bars separate the factors. The LZ78 factorization is output as tuples: `(0,a)(1,a)(0,b)(1,b)(2,a)(3,a)(4,a)(6,$)`. This output is represented by the left trie (a). The LZ78U factorization of the same text is `a|aa|ba|baa|aba|ababa|$`. We output it as `(0,a)(1,a)(0,ba)(3,a)(1,ba)(5,ba)(0,$)`. This output induces the right tree (b).

that is a prefix of F_x, and S_x is the suffix determined by the factorization. We call y the ***referred index*** and S_x the ***factor label*** of the x-th factor. Transforming the factors to this output induces a dictionary tree, called the ***LZ78U-tree***, in which

- every node corresponds to a factor,
- the parent of a node v corresponds to the referred index of v, and
- the edge between the node of the x-th factor and its parent is labeled with the factor label of the x-th factor.

Figure 5 shows a comparison to the LZ78-trie. By the definition of the factorizations, the LZ78-trie is a subtree of the suffix *trie*, whereas the LZ78U-tree is a subtree of the suffix *tree*. The latter can be seen by the fact that the suffix tree compacts the unary paths of the suffix trie. This fact is the foundation of the algorithm we present in the following. It builds the LZ78U-tree on top of the suffix tree. The algorithm is an easier computable variant of the LZ78 algorithms in [10, 19].

The Algorithm. The internal suffix tree nodes can be mapped to the pre-order numbers $[1..n]$ injectively by using rank/select data structures on the suffix tree topology. This allows us to use $n \lg n$ bits for storing a factor id in each internal suffix tree node. To this end, we create an array R of $n \lg n$ bits. All elements of the array are initially set to zero. In order to compute the factorization, we scan the text from left to right. Given that we are at text position i, we locate the suffix tree leaf $\ell \leftarrow \text{leaf-select}(i)$ corresponding to the i-th suffix. Let $p \leftarrow \text{parent}(\ell)$ be ℓ's parent.

- If $R[p] \neq 0$, then p corresponds to a factor F_x. Let c be the first character of the edge label $\lambda(p, \ell)$. The substring $F_x c$ occurs exactly once in T, otherwise ℓ would not be a leaf. Consequently, we output a factor consisting of the referred index $R[p]$ and the string label c. We further increment i by the string depth of p plus one.
- Otherwise, using level ancestor queries, we search for the highest node $v \leftarrow \text{level-anc}(\ell, d)$ with $R[v] = 0$ on the path between the root (exclusively) and p (iterate over the depth d starting with zero). We set $R[v] \leftarrow z + 1$, where z is the current number of computed factors. We output the referred index $R[\text{parent}(v)]$ and the string $\lambda(\text{parent}(v), v)$. Finally, we increment i by the string depth of v.

Table 1 Datasets of size 200MiB. The alphabet size σ includes the terminating $-character. The expression avg_{LCP} is the average of all LCP values. z is the number of LZ77 factors with $t = 1$. The number of runs consisting of one character in BWT is called bwt-runs. H_k denotes the k-th order empirical entropy.

collection	σ	max lcp	avg_{LCP}	bwt-runs	z	$\max_x \lvert F_x \rvert$	H_0	H_3
HASHTAG	179	54,075	84	63,014K	13,721K	54,056	4.59	2.46
PC-DBLP.XML	97	1084	44	29,585K	7035K	1060	5.26	1.43
PC-DNA	17	97,979	60	128,863K	13,970K	97,966	1.97	1.92
PC-ENGLISH	226	987,770	9390	72,032K	13,971K	987,766	4.52	2.42
PC-PROTEINS	26	45,704	278	108,459K	20,875K	45,703	4.20	4.07
PCR-CERE	6	175,655	3541	10,422K	1447K	175,643	2.19	1.80
PCR-EINSTEIN.EN	125	935,920	45,983	153K	496K	906,995	4.92	1.63
PCR-KERNEL	161	2,755,550	149,872	2718K	775K	2,755,550	5.38	2.05
PCR-PARA	6	72,544	2268	13,576K	1927K	70,680	2.12	1.87
PC-SOURCES	231	307,871	373	47,651K	11,542K	307,871	5.47	2.34
TAGME	206	1281	26	65,195K	13,841K	1279	4.90	2.60
WIKI-ALL-VITAL	205	8607	15	80,609K	16,274K	8607	4.56	2.45
COMMONCRAWL	115	246,266	1727	45,899K	10,791K	246,266	5.37	2.78

Since level ancestor queries can be answered in constant time, we can compute a factor in time linear to its length. Summing over all factors we get linear time overall. We use $n \lg n + |\mathsf{ST}|$ bits of working space.

Improved Compression Ratio. To achieve an improved compression ratio, we factorize the factor labels: If S_x is the label of the x-th factor f_x, then we factorize $S_x = G_1 \cdots G_m$ with $G_j := \text{argmax}_{S \in \{F_y : y < x, |F_y| \geq t\} \cup \Sigma} |S|$ greedily chosen for ascending values of j with $1 \leq j \leq m$, with a threshold $t \geq 1$. By doing so, the string S_x gets partitioned into characters and former factors longer than t. The factorization of S_x is done in $\mathcal{O}(|S_x|)$ time by traversing the suffix tree with level ancestor queries, as above (the only difference is that we do not introduce a new factor to the LZ78U factorization).

4 Practical Evaluation

Table 1 shows the text collections used for the evaluation in the tudocomp benchmarks. We provide a tool that automatically downloads and prepares a superset of the collections used in this evaluation. The collections with the prefixes PC or PCR belong to the Pizza&Chili Corpus[6]. The Pizza&Chili Corpus is divided in a real text corpus (PC), and in a repetitive corpus (PCR). The collection HASHTAG is a tab-separated values file with five columns (integer values, a hashtag and a title) [9]. The collection TAGME is a list of Wikipedia fragments[7]. Finally, we present two new text collections. The first collection, called WIKI-ALL-VITAL, consists of the approx. 10,000 most vital Wikipedia articles[8]. We gathered all articles and processed them with the Wikipedia extractor of TANL [24] to convert each article into plain

[6] http://pizzachili.dcc.uchile.cl
[7] http://acube.di.unipi.it/tagme-dataset
[8] https://en.wikipedia.org/wiki/Wikipedia:Vital_articles/Expanded

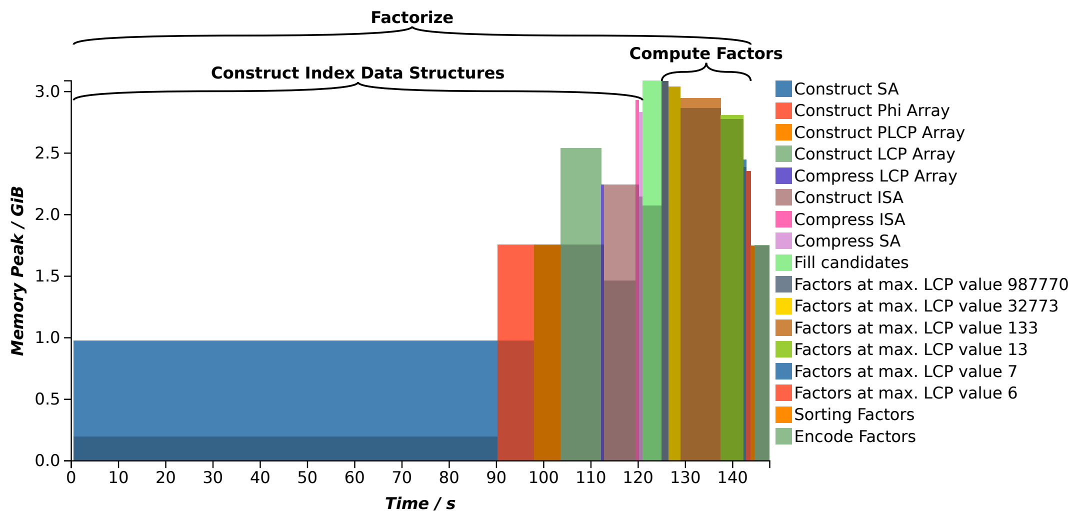


Figure 6 Compression of the collection PC-ENGLISH with `lcpcomp(coder=sle, threshold=5, comp=arrays)`. SA and LCP are built in `delayed` mode. Each phase of the algorithm (like the construction of SA) is depicted as a bar in the diagram. Each bar is additionally highlighted in a different color with a light and a dark shade. The darker part of a phase's bar is the amount of memory already reserved when entering the phase; the lighter part shows the memory peak on top of the already reserved space of the current phase. The memory consumption of a phase on termination is equal to the darker bar of the next phase. Coherent phases are grouped together by curly braces on the top.

text. The second collection, named COMMONCRAWL, is composed of a random subset of a web crawl[9]; this subset contains only the plain texts (i.e., without header and HTML tags) of web sites with ASCII characters.

Setup. The experiments were conducted on a machine with 32 GB of RAM, an Intel Xeon CPU `E3-1271 v3` and a Samsung SSD `850 EVO 250GB`. The operating system was a 64-bit version of Ubuntu Linux 14.04 with the kernel version 3.13. We used a single execution thread for the experiments. The source code was compiled using the GNU compiler `g++ 6.2.0` with the compile flags `-O3 -march=native -DNDEBUG`.

lcpcomp Strategies. For lcpcomp we use the heap strategy and the list decompression strategy described in Section 3.2. We call them `heap` and `compact`, respectively. The strategies described in Section 3.2.2 are called `arrays` (compression) and `scan` (decompression). The decompression strategy `scan` takes the number of scans b as an argument. We encode the remaining substrings of lcpcomp with a static low entropy encoder `sle`. The coder is similar to a Huffman coder, but it additionally treats all 3-grams of the remaining substrings as symbols of the input. We evaluated lcpcomp only with the coder `sle`, since it provided the best compression ratio. We produced SA, ISA and LCP in the `delayed` mode.

LZ78U Implementation. We used the suffix tree implementation `cst_sada` of SDSL, since it provides all required operations like level ancestor queries.

[9] `http://commoncrawl.org`

Table 2 Output of the comparison tool for the collection PCR-CERE. `C` and `D` denote the compression and decompression phase, respectively. `b` and `t` are the parameters b and t, respectively. The tool checks at the last column whether the `sha256`-checksum of the decompressed output matches the input file.

pcr_cere.200MB (200.0MiB, sha256=577486b84633ebc71a8ca4af971eaa4e6a91bcddda17f0464ff79038cf928eab)

Compressor	C Time	C Memory	C Rate	D Time	D Memory	chk
lz78u(t=5,huff)	280.2s	9.2GiB	12.4643%	5.1s	286.9MiB	OK
lcpcomp(t=5,heap,compact)	235.5s	3.4GiB	2.8436%	36.4s	7.6GiB	OK
lcpcomp(t=5,arrays,compact)	103.1s	3.2GiB	2.8505%	36.6s	7.6GiB	OK
lcpcomp(t=5,arrays,scans(b=25))	104.6s	3.2GiB	2.8505%	37.2s	4.6GiB	OK
lzss_lcp(t=5,bit)	98.5s	2.9GiB	4.0530%	4.3s	230.6MiB	OK
code2	16.4s	230.6MiB	28.4704%	6.6s	30.6MiB	OK
huff	2.7s	230.5MiB	28.1072%	5.9s	30.6MiB	OK
lzw	14.3s	480.9MiB	23.4411%	5.5s	452.6MiB	OK
lz78	13.6s	480.8MiB	29.1033%	10.3s	142.9MiB	OK
bwtzip	83.6s	1.7GiB	6.8688%	22.6s	1.4GiB	OK
gzip -1	2.6s	6.6MiB	30.7312%	1.4s	6.6MiB	OK
gzip -9	107.6s	6.6MiB	26.2159%	1.0s	6.6MiB	OK
bzip2 -1	13.1s	9.3MiB	25.3806%	5.1s	8.6MiB	OK
bzip2 -9	13.8s	15.4MiB	25.2368%	5.6s	11.7MiB	OK
lzma -1	12.6s	27.2MiB	27.6205%	3.4s	19.7MiB	OK
lzma -9	138.6s	691.7MiB	1.9047%	337.3ms	82.7MiB	OK

Figure 6 visualizes the execution of lcpcomp with the strategy `arrays` in different phases for the collection PC-ENGLISH. The figure is generated with the JSON output of `tdc` by the chart visualization application on our website `http://tudocomp.org/charter`. We loaded the text (200MiB), constructed SA (800MiB, 32 bits per entry), computed LCP (500MiB, 20-bits per entry), computed ISA (700MiB, 28 bits per entry), and shrunk SA to 700MiB. Summing these memory sizes gives a memory offset of 1.9GiB when lcpcomp started its actual factorization. The factorization is divided in LCP value ranges. After the factorization, the factors were sorted and finally transformed to a binary bit sequence by `sle`. Most of the running time was spent on building SA, roughly 1GiB was spent for creating the lists L_i containing the suffix array entries with an LCP value of i.

Finally, we compare the implemented algorithms of tudocomp with some classic compression programs like `gzip` by our comparison tool `compare.py`. The output of the tool is shown in Table 2. The compressor `lzss_lcp` computes the LZ77 factorization (Def. 1) by a variant of [16]. The compressor `bwtzip` is an alias for the compression pipeline `bwt:rle:mtf:encode(huff)` devised in Section 2.1. The programs `bzip2` and `gzip` do not compress the highly repetitive collection PCR-CERE as well as any of the tudocomp compressors (excluding the plain usage of a coder). Still, our algorithms are inferior to `lzma -9` in the compression ratio and the decompression speed. The high memory consumption of `LZ78U` is mainly due to the usage of the compressed suffix tree.

5 Conclusions

The framework tudocomp consists of a compression library, the command line executable `tdc`, a comparison tool, and a visualization tool. The library provides classic compressors and standard coders to facilitate building a compressor, or constructing a complex compression pipeline. Since the library was built with a focus on high modularity, a compression pipeline does not have to get statically compiled. Instead, the tool `tdc` can assemble a compression pipeline at runtime. Such a pipeline, given as a parameter to `tdc`, can be adjusted in detail at runtime.

We demonstrated tudocomp's capabilities with the implementation of two new compressors: lcpcomp, a variant of LZ77, and LZ78U, a variant of LZ78. Both new variants show better compression ratios than their respective originals, but have a higher memory consumption and also slower decompression times. Further research is needed to address these issues.

Future Research. The memory footprint of lcpcomp could be dropped by exchanging the array implementations of SA, ISA and LCP with compressed data structures like a compressed suffix array, an inverse suffix array sampling, and a permuted LCP (PLCP) array, respectively. We are currently investigating a variant that only observes the peaks in the PLCP array to compute the same output as lcpcomp. If the number of peaks is π, then this algorithm needs at most $\pi \lg n$ bits on top of SA, ISA and the PLCP array.

We are optimistic that we can improve the compression ratio of our algorithms by adapting sophisticated approaches in how the factors are chosen [1, 7, 8] and how the factors are finally coded [5].

References

1. Jyrki Alakuijala and Zoltan Szabadka. Brotli Compressed Data Format. RFC 7932, 2016.
2. David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
3. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
4. David R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.
5. Jarek Duda, Khalid Tahboub, Neeraj J. Gadgil, and Edward J. Delp. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *Proc. PCS*, pages 65–69. IEEE Computer Society, 2015.
6. Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
7. Andrea Farruggia, Paolo Ferragina, and Rossano Venturini. Bicriteria data compression: Efficient and usable. In *Proc. ESA*, volume 8737 of *LNCS*, pages 406–417. Springer, 2014.
8. Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of Lempel-Ziv compression. *SIAM J. Comput.*, 42(4):1521–1541, 2013.
9. Paolo Ferragina, Francesco Piccinno, and Roberto Santoro. On analyzing hashtags in Twitter. In *Proc. ICWSM*, pages 110–119, 2015.
10. Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel-Ziv computation in small space (LZ-CISS). In *Proc. CPM*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, first edition, 1995.
12. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014.
13. Roberto Grossi and Giuseppe Ottaviano. Design of practical succinct data structures for large data collections. In *Proc. SEA*, volume 7933 of *LNCS*, pages 5–17. Springer, 2013.
14. Jan Holub, Jakub Reznicek, and Filip Simek. Lossless data compression testbed: ExCom and Prague corpus. In *Proc. DCC*, page 457. IEEE Computer Society, 2011.
15. Guy Joseph Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

16 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. CPM*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013.

17 Juha Kärkkäinen, Giovanni Manzini, and Simon John Puglisi. Permuted longest-common-prefix array. In *Proc. CPM*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.

18 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):1–19, 2006.

19 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC*, pages 3–12. IEEE Computer Society, 2016.

20 N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proc. DCC*, pages 296–305. IEEE Computer Society, 1999.

21 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

22 Wataru Matsubara, Kazuhiko Kusano, Hideo Bannai, and Ayumi Shinohara. A series of run-rich strings. In *Proc. LATA*, volume 5457 of *LNCS*, pages 578–587. Springer, 2009.

23 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

24 Maria Simi and Giuseppe Attardi. Adapting the tanl tool suite to universal dependencies. In *Proc. LREC*. European Language Resources Association, 2016.

25 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

26 Peter Weiner. Linear pattern matching algorithms. In *Proc. Annual Symp. on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, 1973.

27 Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

28 Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999.

29 Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.

30 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, 23(3):337–343, 1977.

31 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.

A Cycle-Free Lemma of lcpcomp

▶ **Lemma 4.** *The output of lcpcomp contains enough information to restore the original text.*

Proof. We want to show that the output is free of cycles, i.e., there is no text position i for that $i \underbrace{\rightarrow \cdots \rightarrow}_{\text{cycle length}} i$ holds, where $\rightarrow$ is a relation on text positions such that $i \rightarrow j$ holds iff there is a substring $T[i'..i'+\ell-1]$ with $i \in [i', i'+\ell-1]$ that has been substituted by a reference $(j-i+i', \ell)$. If the text is free of cycles, then each substituted text position can be restored by following a finite chain of references.

First, we show that is not possible to create cycles of length two. Assume that we substituted $T[\mathsf{SA}[i]\,..\,\mathsf{SA}[i]+\ell_i-1]$ with $(\mathsf{SA}[i-1]\,, \ell_i)$ for $t \leq \ell_i \leq \mathsf{LCP}[i]$. The algorithm will not choose $T[\mathsf{SA}[i-1]+k..\,\mathsf{SA}[i-1]+k+\ell_k-1]$ for $0 \leq k \leq \ell_i$ and $t \leq \ell_k \leq \mathsf{LCP}[i]-k$ to be substituted with $(\mathsf{SA}[i]+k, \ell_k)$, since $T[\mathsf{SA}[i]+k..] > T[\mathsf{SA}[i-1]+k..]$ and therefore $\mathsf{ISA}[\mathsf{SA}[i]+k] > \mathsf{ISA}[\mathsf{SA}[i-1]+k]$. Finally, by the transitivity of the lexicographic order (i.e., the order induced by the suffix array), it is neither possible to produce larger cycles. ◀

Table 3 Compression and decompression with the lcpcomp strategies `arrays` and `scan`, for fixed parameters t and b. For each collection we chose the t with the best compression ratio. Having t fixed, we chose the $b \le 40$ with the shortest decompression running time.

	compression					decompression		
collection	t	#factors	ratio	memory	time	b	memory	time
HASHTAG	5	10,088,662	25.47%	3179.9	100	17	1726	50
PC-DBLP.XML	5	5,547,102	14.4 %	2929.7	99	28	1993.5	65
PC-DNA	21	1,091,010	26.03%	2925	122	11	291.2	8
PC-ENGLISH	5	11,405,635	27.66%	3162	123	25	792.6	36
PC-PROTEINS	10	1,749,917	35.91%	2900	124	13	362	11
PCR-CERE	22	236,551	2.45 %	3126	113	6	454.2	7
PCR-EINSTEIN.EN	8	24,672	0.1 %	3288.8	113	40	1777.3	47
PCR-KERNEL	6	512,047	1.51 %	3356.3	116	40	2129.6	37
PCR-PARA	22	388,195	3.27 %	3060.8	117	6	402.3	7
PC-SOURCES	5	8,922,703	23.36%	3271	98	30	1019.6	36
TAGME	5	10,986,096	27.29%	2987.7	113	25	985.4	41
WIKI-ALL-VITAL	5	13,338,470	32.46%	3163	117	27	870.4	45
COMMONCRAWL	4	8,402,041	21.49%	3254.6	101	36	1206.11	41

B LZ78U Offline Algorithm

Instead of directly constructing the array R that is necessary to determine the referred indices, we create a list F storing the marked LZ-trie nodes, and a bit vector B marking the internal nodes belonging to the LZ-tree. Initially, only the root node is marked in B. Let i, p and ℓ be defined as in the above tree traversal. If $B[p]$ is set, then we append ℓ to F and increment i by one. Otherwise, by using level ancestor queries, we search for the highest node v with $B[v] = 0$ on the path between the root and p. We set $B[v] \leftarrow 1$, and append v to F. Additionally, we increment i by $|\lambda(\text{parent}(v), v)|$. By doing so, we have computed the factorization.

In order to generate the final output, we augment B with a rank data structure, and create a permutation N that maps a marked suffix tree node to the factor it belongs. The permutation N is represented as an array of $z \lg z$ bits, where $N[B.\text{rank}_1(F[x])] \leftarrow x$, for $1 \le x \le z$. At this point, we no longer need F. The rest of the algorithm sorts the factors in the factor index order. To this end, we create an array R with $z \lg z$ bits to store the referred indices, and an array S with $z \lg n$ bits to store the factor labels. To compute S and R, we scan all marked nodes in B: Since the x-th marked node v corresponds to the $N[x]$-th factor, we can fill up S easily: If v is a leaf, we store the first character of $\lambda(\text{parent}(v), v)$ in $S[N[x]]$; otherwise (v is an internal node), we store the whole string. Filling R is also easy if v is a child of the root: we simply store the referred index 0. Otherwise, the parent p of v is not the root; p corresponds to the y-th factor, where $y := N[B.\text{rank}_1(p)]$.

The algorithm using $|\mathsf{ST}| + n + z(\lg(2n) + \lg z) + 2z \lg n + o(n)$ bits of working space, and runs in linear time.

C LZ78U Code Snippet

```
void factorize(TextDS<>& T, SuffixTree& ST, std::function<void(size_t begin, size_t end,
    size_t ref)> output){
 typedef SuffixTree::node_type node_t;
 sdsl::int_vector<> R(ST.internal_nodes,0,bits_for(T.size() * bits_for(ST.cst.csa.sigma) /
    bits_for(T.size())));
 size_t pos = 0, z = 0;
 while(pos < T.size() - 1) {
  node_t l = ST.select_leaf(ST.cst.csa.isa[pos]);
  size_t leaflabel = pos;
  if(ST.parent(l) == ST.root || R[ST.nid(ST.parent(l))] != 0) {
   size_t parent_strdepth = ST.str_depth(ST.parent(l));
   output(pos + parent_strdepth, pos + parent_strdepth + 1, R[ST.nid(ST.parent(l))]);
   pos += parent_strdepth+1;
   ++z;
   continue;
  }
  size_t d = 1;
  node_t parent = ST.root;
  node_t node = ST.level_anc(l, d);
  while(R[ST.nid(node)] != 0) {
   parent = node;
   node = ST.level_anc(l, ++d);
  }
  pos += ST.str_depth(parent);
  size_t begin = leaflabel + ST.str_depth(parent);
  size_t end = leaflabel + ST.str_depth(node);
  output(begin, end, R[ST.nid(ST.parent(node))]);
  R[ST.nid(node)] = ++z;
  pos += end - begin;
 }
}
```

Figure 7 Implementation of the LZ78U algorithm streaming the output

D More Evaluation

In this section, the execution time is measured in second, and all data sizes are measured in mebibytes (MiB). In Table 3, we selected the t with the best compression ratio and the b with the shortest decompression time. Although t and b tend to correlate with the compression speed and decompression memory, respectively, selecting values for t and b that yield a good compression ratio or a fast decompression speed seems difficult.

In Table 4, we fixed two values of t and three values of b. The compression ratio of the strategies `heap` and `arrays` differ slightly, since the lcpcomp compression scheme does not specify a tie breaking rule for choosing a longest repeated substring.

Figure 8 compares the number of factors of `lzss_lcp` with `lcpcomp`'s `arrays` strategy on all aforementioned datasets. We varied the threshold t from 4 up to 22 and measured for each t the number of created factors. In all cases, lcpcomp produces less factors than `lzss_lcp` with the same threshold.

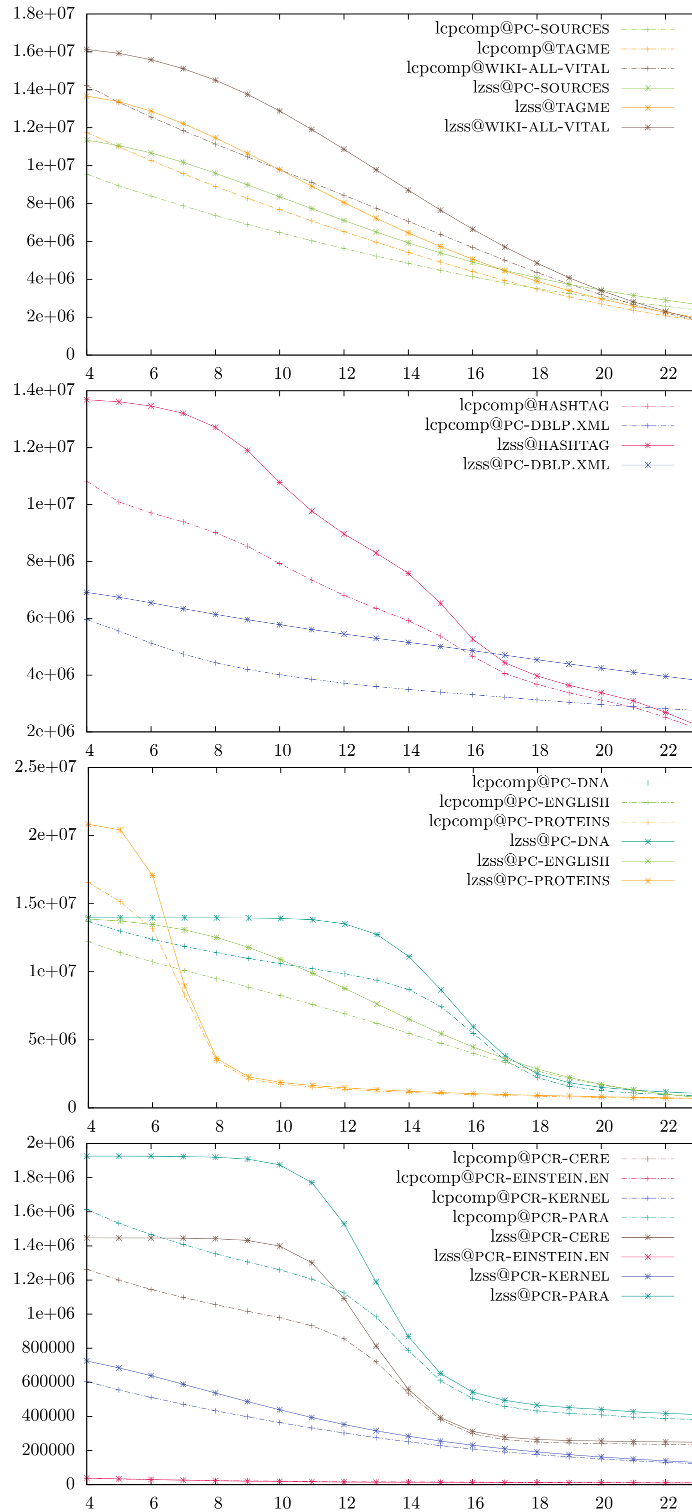


Figure 8 Number of factors (y-axis) of lcpcomp and LZ77 on varying the given threshold t (x-axis).

Table 4 Evaluation of external compression programs and algorithms of the tudocomp framework on the collection COMMONCRAWL.

	compression			decompression		
compressor	memory	output size	time	strategy	memory	time
external programs						
`gzip -1`	6.6	61.3	2.19		6.6	1.045
`bzip2 -1`	9.3	55.4	14.455		8.6	4.7
`lzma -1`	27.2	46.7	9.395		19.7	2.37
`gzip -9`	6.6	53.4	6.86		6.6	0.97
`bzip2 -9`	15.4	50.7	14.78		11.7	4.955
`lzma -9e`	691.7	29.4	104.375		82.7	1.56
tudocomp algorithms						
`encode(sle)`	265.2	137.7	24.145		30.6	10.095
`encode(huff)`	230.4	135	5.7		30.4	9.045
`bwtzip`	1730.6	43.7	83.035		1575	21.44
`lcpcomp`($t = 5$,`heap`)	3598.9	44.1	228.055	`compact`	6592.2	33.24
`lcpcomp`($t = 22$,`heap`)	3161.7	58.5	175.21	`compact`	3981.2	14.065
`lcpcomp`($t = 5$,`arrays`)	3354.2	44.3	107.34	`scan`($b = 6$)	4930	43.1
				`scan`($b = 25$)	2584.5	33.995
				`scan`($b = 60$)	1164.8	38.925
`lcpcomp`($t = 22$,`arrays`)	2980.6	58.5	109.245	`scan`($b = 6$)	1308	10.925
				`scan`($b = 25$)	520.9	11.265
				`scan`($b = 60$)	368.7	15.635
`lzss(bit)`	2980.4	60.2	108.59		230.6	6.045
`lz78(bit)`	480.8	83.1	17.96		254.9	11.46
`lzw(bit)`	480.8	70.3	18.97		663.1	7.05

E LZ78U Pseudo Codes

Algorithm 1: Streaming LZ78U

```
1  ST ← suffix tree of T
2  R ← array of size n                 // maps internal suffix tree nodes to LZ trie ids
3  initialize R with zeros
4  pos ← 1                                                          // text position
5  z ← 0                                                        // number of factors
6  while pos ≤ |T| do
7      ℓ ← leaf-select(ISA[pos])
8      if R[parent(ℓ)] ≠ 0 or parent(ℓ) = root then
9          output the first character of λ(parent(ℓ), ℓ)
10         output referred index R[parent(node)]
11         z ← z + 1
12         pos ← pos + str_depth(parent) + 1
13     else
14         d ← 1                                                // the current depth
15         while R[level-anc(ℓ, d)] ≠ 0 do
16             d ← d + 1
17             pos ← pos + |λ(level-anc(ℓ, d − 1), level-anc(ℓ, d))|
18         node ← level-anc(ℓ, d)
19         z ← z + 1
20         R[node] ← z
21         output string λ(parent(node), node)
22         output referred index R[parent(node)]
23         pos ← pos + |λ(parent(node), node)|
```

Algorithm 2: Computing LZ78U memory-efficiently

```
1  ST ← suffix tree of T
2  pos ← 1
3  B ← bit vector of size n            // marking the ST nodes belonging to the LZ-trie
4  F ← list of integers  // storing the LZ-trie nodes in the order when they got explored
5  node ← root of ST
6  while pos ≤ |T| do
7      node ← child(node, T[pos])                   // use level-anc to get O(1) time
8      pos ← pos + (is-leaf(node) ? 1 : |λ(parent(node), node)|
9      if is-leaf(node) or B[node] = 0 then
10         B[node] ← 1
11         F.append(node)
12         node ← root of ST
13 add_rank_support(B)
14 N ← array of length z    // stores for each marked ST node to which factor it belongs
15 for 1 ≤ x ≤ z do N[B.rank_1(F[x])] ← x
16 F ← integer array of size z                          // storing the referred indices
17 S ← string array of size z                       // storing the string of each factor
18 for 1 ≤ x ≤ z do
19     node ← B.rank_1(x)
20     if is-leaf(node) then S[N[x]] ← first character of λ(parent(node), node)
21     else S[N[x]] ← λ(parent(node), node)
22     if parent(node) = root then F[N[x]] ← 0
23     else F[N[x]] ← N[B.rank_1(parent(node))]
24 return (F,S)
```

Algorithm Engineering for All-Pairs Suffix-Prefix Matching*

Jihyuk Lim[1] and Kunsoo Park[†2]

1 **Department of Computer Science and Engineering, Seoul National University, Seoul, Korea**
`jhlim@theory.snu.ac.kr`

2 **Department of Computer Science and Engineering, Seoul National University, Seoul, Korea**
`kpark@theory.snu.ac.kr`

Abstract

All-pairs suffix-prefix matching is an important part of DNA sequence assembly where it is the most time-consuming part of the whole assembly. Although there are algorithms for all-pairs suffix-prefix matching which are optimal in the asymptotic time complexity, they are slower than SOF and Readjoiner which are state-of-the-art algorithms used in practice. In this paper we present an algorithm for all-pairs suffix-prefix matching that uses a simple data structure for storing input strings and advanced algorithmic techniques for matching, which together lead to fast running time in practice. Our algorithm is 14 times faster than SOF and 18 times faster than Readjoiner on average in real datasets and random datasets.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases all-pairs suffix-prefix matching, algorithm engineering, DNA sequence assembly

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.14

1 Introduction

The problem of all-pairs suffix-prefix (APSP) matching is defined as follows: Given a collection of k strings $S_1, S_2, \ldots, S_k$, find the longest suffix of S_i which is a prefix of S_j for all pairs S_i and S_j. Let N be the sum of lengths of the input strings $S_1, S_2, \ldots, S_k$. All-pairs suffix-prefix matching is an important part of DNA sequence assembly where it is the most time-consuming part of the whole assembly process. In DNA sequence assembly, a parameter om (for overlap minimum) is given for APSP matching where we want to find the longest overlap of S_i and S_j whose length is at least om. The output of APSP matching can be stored in a $k \times k$ matrix Ov, where $Ov[i, j]$ is the length of the longest suffix of S_i that is a prefix of S_j. Alternatively, the output can be a list of three integers $(i, j, Ov[i, j])$ such that $Ov[i, j] \geq om$ as a compact representation.

All-pairs suffix-prefix matching has been studied in the fields of stringology and bioinformatics. In general, a solution for APSP matching consists of two phases: the first phase is to build a data structure which represents all prefixes of the input strings, and the second

* This research was supported by Collaborative Genome Program for Fostering New Post-Genome industry through the National Research Foundation of Korea(NRF) funded by the Ministry of Science ICT and Future Planning (No. NRF-2014M3C9A3063541).

† Corresponding author.

© Jihyuk Lim and Kunsoo Park;
licensed under Creative Commons License CC-BY

16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 14; pp. 14:1–14:12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

matching phase is to search the data structure to find occurrences of suffixes of each input string (or equivalently, one can build a data structure representing all suffixes of the input strings, and search it for prefixes of each string). Gusfield et al. [8] proposed a novel algorithm of optimal $O(N + k^2)$ time for APSP matching by building a generalized suffix tree for the input strings. Ohlebusch and Gog [16] gave another $O(N + k^2)$-time algorithm by building an enhanced suffix array [1, 14] for the input strings, which improves upon Gusfield et al.'s in running time and space. Tustumi et al. [19] further improved the running time and space of Ohlebusch and Gog's algorithm. Louza et al. [13] presented a parallel algorithm for APSP matching which is based on Tustumi et al.'s. In bioinformatics too, many algorithms have been proposed for APSP matching [5, 6, 11, 15, 9, 10, 18, 17], and Readjoiner [6] and SOF [9] are state-of-the-art algorithms which show best performances in practice. Although the algorithms in [8, 16, 19] are optimal in the asymptotic worst-case time complexity, Readjoiner and SOF are faster than these algorithms in practice. Hence there is a mismatch between theoretical results and practice. A main reason for the mismatch is that the generalized suffix tree [8, 7] and even the enhanced suffix array [1, 14] are heavy machineries (though they provide powerful functionalities) and so the constants hidden in the asymptotic notations are quite big. On the other hand, SOF uses a simple but effective data structure called the compact prefix tree (also known as compact trie) for the input strings and its matching phase uses quite naive algorithmic techniques. As another approach, we can build the Aho-Corasick automaton [2] for the input strings, and solve APSP matching by searching the automaton for each input string. We implemented this approach, but the Aho-Corasick automaton is another piece of heavy machinery and just building it (without the matching phase) takes more time than the whole SOF.

In this paper we propose a fast algorithm for APSP matching. We first build a compact prefix tree for the input strings, but in the matching phase we need more advanced techniques because the only functionality that the compact prefix tree provides is to check whether a given string is a prefix of the input strings or not. We divide the matching phase into three cases depending on the lengths of suffixes of an input string, and in each case we use an appropriate algorithmic technique which finds efficiently occurrences of the suffixes of an input string corresponding to the case. We did experiments to compare our algorithm against SOF and Readjoiner with real datasets and random datasets. In the experiments our algorithm is 14 times faster than SOF and 18 times faster than Readjoiner on average. We also obtain reasonable scalability with a parallel implementation of our algorithm.

2 New algorithm for APSP matching

Let S be a string of length n over an alphabet Σ. We denote the length of S by $|S|$. The i-th character of S is denoted by $S[i]$ $(1 \leq i \leq |S|)$, and a substring $S[i]S[i+1]...S[j]$ by $S[i..j]$. A substring $S[1..i]$ for $1 \leq i \leq n$ is called a prefix of S and a substring $S[i..n]$ for $1 \leq i \leq n$ is called a suffix of S. For strings A and B, we use $A \prec B$ to denote that A is lexicographically smaller than B.

We describe the compact prefix tree CT of k input strings defined in [9]. The compact prefix tree is basically a compact trie, i.e., there is one leaf corresponding to each input string and every internal node has at least two children. An array *sorted* stores the lexicographic ordering of the input strings such that $S_{sorted[1]} \preceq S_{sorted[2]} \preceq ... \preceq S_{sorted[k]}$. The leaves of CT are in the lexicographic order, and each node v of CT has an interval $[a, b]$ such that $S_{sorted[a]}, \ldots, S_{sorted[b]}$ are the leaves in the subtree rooted at v. The compact prefix tree provides a function Find(s), which returns the node v nearest to the root such that s is a

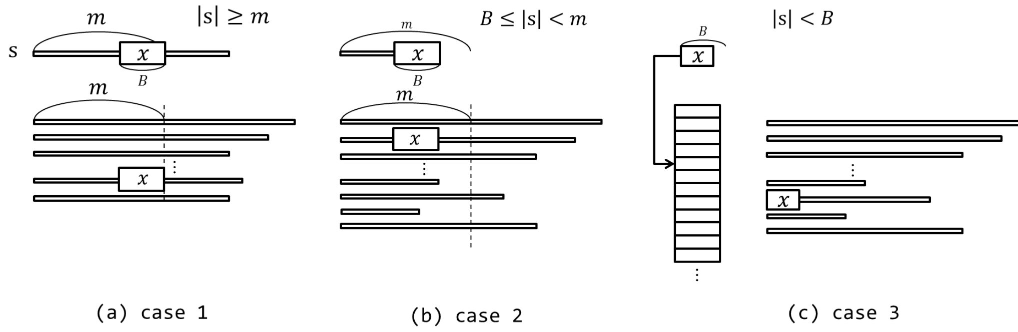


Figure 1 Three cases of the matching step.

prefix of the string on the path from the root to v. If such a node does not exist, Find(s) returns NULL. For notational convenience, let *isort* be the inverse function of *sorted*, i.e., $isort[j] = c$ if $j = sorted[c]$.

We first describe an overview of our algorithm for APSP matching. Our algorithm consists of three steps: preprocessing, matching, and output steps. Our algorithm uses two integers m and B as parameters ($m \geq B$).

- In the preprocessing step we construct a compact prefix tree CT for k input strings as in [9]. In addition, we build auxiliary data structures which will be used in the matching step.
- In the matching step we consider each input string S_i separately and do the following. For each suffix s of S_i, we find the interval $[a, b]$ such that $S_{sorted[a]}, \ldots, S_{sorted[b]}$ have s as their prefixes. If the interval $[a, b]$ is not empty, we insert $(a, b, |s|)$ into $oList[i]$. But if we find the interval $[a, b]$ by calling Find(s) for every suffix s of S_i, it will take too much time. We reduce the number of calls to Find by dividing the suffixes of S_i into three cases and using different techniques for the cases. Figure 1 illustrates three cases of the matching step.
  - In case 1, we consider each suffix s of S_i such that $|s| \geq m$. If $s[m-B+1..m]$ appears in $S_j[m-B+1..m]$ for some j (see Figure 1 (a)), we call Find(s); otherwise, it is guaranteed that s does not appear as a prefix of input strings and so we don't need to call Find(s).
  - In case 2, we consider each suffix s such that $B \leq |s| < m$. If $s[|s|-B+1..|s|]$ appears in $S_j[|s|-B+1..|s|]$ for some j (see Figure 1 (b)), we call Find(s); otherwise, s does not appear as a prefix of input strings.
  - In case 3, we consider suffixes s such that $|s| < B$. For this case we precompute Find(s') for every string s' of length less than B which appears as a prefix of input strings, and we store them in a table *Bprefix*. Hence there are no calls to Find during the matching step.
- In the output step we find the longest overlap of S_i and S_j for every j, which is the largest l such that (a, b, l) is in $oList[i]$ and interval $[a, b]$ contains $isort[j]$.

2.1 Preprocessing step

In the preprocessing step, we build data structures: a compact prefix tree CT, *sorted*, and auxiliary data structures *qrm*, *qList*, and *Bprefix*.

We construct a compact prefix tree CT by inserting input strings one by one into the tree while maintaining the lexicographic order of characters for children of each internal node. At the end of the insertions, then, the input strings appear in the leaves of CT in the lexicographic order. Hence *sorted* and the interval $[a, b]$ of each node can be obtained by traversing the tree.

The values of two parameters m and B are set as follows. Let m_s be the length of a shortest string among k input strings. We define m as follows.

$$m = \begin{cases} m_s & \text{if } \frac{N}{c_1 k} \leq m_s \leq \frac{N}{c_2 k} \\ \frac{N}{c_1 k} & \text{if } m_s < \frac{N}{c_1 k} \\ \frac{N}{c_2 k} & \text{if } m_s > \frac{N}{c_2 k}, \end{cases} \tag{1}$$

for some constants c_1 and c_2. We define $B = \min(\log_{|\Sigma|} 2mk, m)$, and a string of length B will be called a *block*.

We will use an auxiliary data structure qrm for case 1 in Figure 1. Let $m' = \max(m, om)$, and let k' be the number of input strings whose length is at least m'. Let $\mathcal{S}' = \{S'_1, S'_2, \ldots, S'_{k'}\}$ be the collection of the length-m prefixes of input strings whose length is at least m'. Hence $\mathcal{S}'$ looks like a $k' \times m$ matrix as in Figure 1 (a). For every string x of length B, $qrm[f(x)]$ is the rightmost occurrence of x in $\mathcal{S}'$, i.e.,

$$qrm[f(x)] = \begin{cases} \max\{q \mid x = S'_i[q - B + 1..q] \text{ for } 1 \leq i \leq k'\} & \text{if } x \text{ appear in } \mathcal{S}' \\ B - 1 & \text{otherwise}, \end{cases} \tag{2}$$

where $f(x)$ is a function mapping a string x to an integer used as an index of the qrm table, i.e.,

$$f(x) = \sum_{i=1}^{|x|} rank(x[i])|\Sigma|^{i-1}, \tag{3}$$

where $rank(c)$ is a function mapping a character c to a lexicographic order of c within the range $[0,\ \Sigma - 1]$. To compute the values of qrm, we scan all blocks (i.e., all substrings of length B) in $\mathcal{S}'$. The table qrm is initially set to $B-1$. In each position $q = B, B+1, \ldots, m$, we consider k' blocks $x_i = S'_i[q - B + 1..q]$ for $1 \leq i \leq k'$ and set $qrm[f(x_i)]$ to q.

We will use *qList* for case 2 in Figure 1. Let $B' = \max(B, om)$ and let k'' be the number of input strings whose length is at least B'. Let $\mathcal{S}'' = \{S''_1, S''_2, \ldots, S''_{k''}\}$ be the collection of input strings whose length is at least B'. For the last block x of S''_j for every $1 \leq j \leq k''$, $qList[f(x)]$ is defined as a list of all distinct positions q such that $S''_i[q - B + 1..q] = x$ for $1 \leq i \leq k''$ and $B' \leq q \leq m$.

If $om \geq m$, we do not compute *qList* because case 2 finds overlaps whose length is less than m. If $om < m$ we compute *qList* by scanning all blocks in $\mathcal{S}''$ as follows. We define a temporary table T that indicates whether a block x appears at the end of some input string. That is, if $|S_j| \geq B$ and $x = S_j[|S_j| - B + 1..|S_j|]$, $T[f(x)] = 1$; otherwise, $T[f(x)] = 0$. In each position q, we consider k'' blocks $x_i = S''_i[q - B + 1..q]$ for $1 \leq i \leq k''$. If $T[f(x_i)] = 1$ and q is not in $qList[f(x_i)]$ (i.e., q is not at the front of $qList[f(x_i)]$), we insert q into $qList[f(x_i)]$.

We will use *Bprefix* for case 3 in Figure 1. Consider a string s such that $|s| < B$. If s is a prefix of some input string (i.e., s appears as a prefix in CT), $Bprefix[f'(s)]$ is a pointer to the node v nearest to the root of CT such that s is a prefix of the string on the path from

the root to v, where $f'(s)$ is a function mapping a string s to an integer:

$$f'(s) = \sum_{i=1}^{|s|-1} |\Sigma|^i + f(s). \tag{4}$$

Note that $f'(s)$ maps a string s into an integer within the range $[0, \sum_{i=1}^{B-1} |\Sigma|^i - 1]$. If s is not a prefix of the input strings, *Bprefix*$[f'(x)]$ is set to NULL.

If $om \geq B$, we do not compute *Bprefix* because case 3 finds overlaps whose length is less than B. If $om < B$, we compute *Bprefix* as follows. Initially, all entries of *Bprefix* are set to NULL. We traverse CT for all character depths (i.e., number of characters on the path from the root) less than B, and set *Bprefix*$[f'(s)]$ for string s that appears as a prefix in CT.

The space complexity of our algorithm is bounded by the memory space used by k input strings and the data structures CT, *sorted*, qrm, $qList$, and *Bprefix*. The input uses $N \log |\Sigma|$ bits (i.e. $O(N)$ space), and the data structures use $O(km)$ space, which is $O(N)$ because $km = \Theta(N)$.

2.2 Matching step

In the matching step we consider each input string S_i separately and we need to find $v = \text{Find}(s)$ for every suffix s of S_i. If v is not NULL, we insert $(a, b, |s|)$ into $oList[i]$, where $[a, b]$ is the interval of v. To reduce the number of calls to Find, we have the three cases in Figure 1.

In case 1, we consider suffixes s of S_i such that $|s| \geq m'$ from longest to shortest. Let p be the start position of a current suffix s to be considered. Initially, $p = 1$.

- If $qrm[f(x)]$ is not m, the current suffix s cannot appear in CT as a prefix and so we don't need to call Find(s). Moreover, suffixes of S_i starting at positions $p+1, p+2, \ldots, m - qrm[f(x)] - 1$ cannot appear in CT as prefixes. Hence p is updated to $p + m - qrm[f(x)]$.
- If $qrm[f(x)]$ is m, we make a call Find(s). If Find returns a node v, we insert $(a, b, |s|)$ into $oList[i]$, where $[a, b]$ is the interval of v. If Find returns NULL, we do nothing. Finally, we increase p by 1.

The preprocessing and matching steps of case 1 are essentially the same as those in Wu and Manber's algorithm [20], which is a Boyer-Moore type algorithm [3, 4, 12], but cases 2 and 3 are different from Wu and Manber's.

In case 2, we consider suffixes s of S_i such that $B' \leq |s| < m'$. If $om \geq m$ (i.e., $m' = \max(m, om) = om$), we skip case 2. In case 2, therefore, $m' = m$. Note that the last blocks of the suffixes considered in this case are $x = S_i[|S_i| - B + 1..|S_i|]$. Since a position q in $qList[f(x)]$ means that x appears at (ending) position q in one of the strings in $\mathcal{S}''$, the length-q suffix s of S_i (i.e. $s = S_i[|S_i| - q + 1..|S_i|]$) may appear in CT. Hence, we make a call Find(s) for every position q in $qList[f(x)]$.

In case 3, we consider suffixes s of S_i such that $|s| < B'$. If $om \geq B$ (i.e., $B' = \max(B, om) = om$), we skip case 3. In case 3, therefore, $B' = B$. For every suffix s of S_i such that $om \leq |s| < B$, we look up *Bprefix*$[f'(s)]$, which already contains the result of Find(s).

The pseudocode of the matching step is shown in Algorithm 1.

2.3 Output step

In the output step we find the longest suffix of S_i that is a prefix of S_j for every j, whose length is the largest l such that tuple (a, b, l) is in $oList[i]$ and interval $[a, b]$ contains $isort[j]$. (We assume that the output of APSP matching is a list of three integers (i, j, l) because it

Algorithm 1 Fast algorithm for APSP matching

```
 1: procedure FastAPSP({S_1, S_2, ..., S_k}, om)
 2:     Precompute m, B, m', B', CT, sorted, qrm, qList and Bprefix
 3:     for i ← 1 to k do                                          ▷ Matching step
 4:         p ← 1                                                        ▷ Case 1
 5:         while p ≤ |S_i| − m' + 1 do
 6:             x ← S_i[p + m − B..p + m − 1]
 7:             if qrm[f(x)] = m then
 8:                 v ← Find(S_i[p..|S_i|])
 9:                 if v ≠ NULL then
10:                     oList[i].insert(v.interval, |S_i| − p + 1)
11:                 p ← p + 1
12:             p ← p + m − qrm[f(x)]
13:         if |S_i| ≥ B' then                                           ▷ Case 2
14:             x ← S_i[|S_i| − B + 1..|S_i|]
15:             for each q in qList[f(x)] do
16:                 v ← Find(S_i[|S_i| − q + 1..|S_i|])
17:                 if v ≠ NULL then
18:                     oList[i].insert(v.interval, q)
19:         for p ← max(|S_i| − B + 2, 1) to |S_i| − om + 1 do           ▷ Case 3
20:             x ← S_i[p..|S_i|]
21:             v ← Prefix[f'(x)]
22:             if v ≠ NULL then
23:                 oList[i].insert(v.interval, |S_i| − p + 1)
24:         Perform output step for oList[i]
```

is a more compact representation in DNA sequence assembly.) In other words, for every $1 \leq c \leq k$ we want to find the largest l such that (a, b, l) is in $oList[i]$ and interval $[a, b]$ contains c. Then l is the length of the largest overlap of S_i and $S_{sorted[c]}$ and thus we output $(i, sorted[c], l)$.

Imagine that intervals $[a, b]$ in $oList[i]$ are on the x-axis. We scan the intervals from $c = 1$ to k, and maintain the values of l in tuples (a, b, l) such that interval $[a, b]$ contains the current c in a max-heap. Then for every current c, we output $(i, sorted[c],$ max value in max-heap$)$. This process can be implemented as follows. We first make an array tA of (a, l)'s and an array tB of (b, l)'s from tuples (a, b, l) in $oList[i]$. We sort tA in non-decreasing order of a's and tB in non-decreasing order of b's. Finally we increase c from 1 to k, and if c hits a of (a, l), we insert l into the max-heap, and if c hits b of (b, l'), we delete l' from the max-heap. Then $(i, sorted[c],$ max value in max-heap$)$ for every current c is a correct output.

Since the number of tuples in $oList[i]$ is at most $|S_i|$, the space usage of $oList[i]$, tA, and tB is $O(|S_i|)$ (thus $O(N)$), and this space for $oList$ can be reused for every $1 \leq i \leq k$.

2.4 Implementation options

The implementation of our algorithm provides several options: *overlap minimum*, *output*, and *parallel* options. The *overlap minimum* option is given by $-om\ i$, where i is the value of overlap minimum om. The *output* option receives 1, 2, or 3 as a parameter. In the case of 1, the program provides the matrix Ov as output. In the case of 2, the program gives the list of

Table 1 Real datasets used in experiments.

	clementina	sinensis	trifoliata	C. elegans	Atta
N	104640576	154995828	46648250	167035020	315387616
k	118365	208909	62344	334465	2835
avg length	884.05	741.93	748.24	499.41	111247.84
m_s	18	13	89	7	1929

three integers $(i, j, Ov[i, j])$. In the case of 3, it gives the list of all overlaps for each pair (not only the longest overlap) like Readjoiner [6] and SOF [9] with $-o$ 2. Our program with option 3 presents all the overlaps by outputting $(i, sorted[a], l), (i, sorted[a+1], l), \ldots, (i, sorted[b], l)$ from each tuple (a, b, l) in $oList[i]$ instead of running the output step. Given the number p of threads as a parameter for the *parallel* option, our program is executed in parallel. In the preprocessing step, qrm and $qList$ are computed in parallel. The matching step is also executed in parallel by p threads.

3 Experiments

Tustumi et al. [19] and Louza et al. [13] compared only the matching phases of their optimal algorithms for APSP matching against SOF and Readjoiner, not accounting for the time to build the data structures to store input strings, and SOF and Readjoiner are in general faster than their algorithms. (If the time to build the data structures is included, the gap would be greater.) Among practical algorithms [5, 6, 11, 15, 9, 10, 18, 17] for APSP matching, SOF and Readjoiner show best performances. Therefore, we compared our algorithm[1] with SOF and Readjoiner. Our algorithm and SOF were compiled with g++ (v. 4.9.2) with the `-O3` optimization flag. Readjoiner (version 1.2) was compiled using the provided Makefile with "64bit=yes assert=no amalgamation=yes threads=yes". For parallel experiments, we used the OpenMP library. All experiments were conducted on a computer with Intel Xeon X5672 CPU, which has 8 cores, 32 GB RAM, and the Linux debian 3.2.0-4-amd64 operating system.

We used two types of datasets which are real and random. The five real datasets are the complete EST databases of *Citrus clementina*[2], *Citrus sinensis*[2], *Citrus trifoliata*[2], C. *elegans*[3], and *Atta cephalotes*[4], which were used as the datasets in the SOF paper [9]. The alphabet of the datasets is $\{A, C, G, T\}$. Table 1 shows specific information of the datasets. Whereas Readjoiner discards low-quality reads before APSP matching, we made three algorithms take all reads as input strings for a fair comparison. The random datasets are generated by a program[1] that gives k strings such that the lengths of the strings follow a normal distribution with mean μ and standard deviation σ and the characters of the strings follow a uniform distribution over the alphabet, where k, μ and σ are parameters given by the user. The alphabet is again $\{A, C, G, T\}$. We generated two datasets $rnd1$ and $rnd2$, where $rnd1$ has 300000, 1000, and 150 as k, μ, and σ, respectively, and $rnd2$ has 1000000, 500, and 100.

We compare the performances of SOF, Readjoiner, and our algorithm for the whole process of APSP matching, i.e., including the time to build their own data structures, the

[1] `http://theory.snu.ac.kr/?p=814`
[2] `http://www.citrusgenomedb.org`
[3] `http://www.uni-ulm.de/in/theo/research/seqana`
[4] `http://antgenomes.org`

Table 2 Running time (in second) of algorithms for real datasets.

om	algorithm	*clementina*	*sinensis*	*trifoliata*	*C. elegans*	*Atta*
10	SOF	22.68	44.18	8.07	62.44	23.91
	Readjoiner	114.13	450.48	57.52	231.66	-
	Our1	3.44	6.10	1.50	14.83	0.70
	Our2	5.07	11.21	1.94	14.52	0.86
15	SOF	21.98	40.60	7.70	61.94	24.58
	Readjoiner	50.47	137.54	33.87	84.25	-
	Our1	2.90	4.90	1.37	14.19	0.64
	Our2	3.36	6.44	1.42	13.84	0.73
20	SOF	20.82	36.60	7.35	60.48	23.79
	Readjoiner	11.53	38.42	3.36	44.15	-
	Our1	2.78	4.33	1.30	14.02	0.64
	Our2	2.94	4.75	1.29	13.67	0.69
25	SOF	20.35	35.89	7.33	59.62	23.86
	Readjoiner	7.26	18.80	3.07	27.37	-
	Our1	2.72	4.29	1.29	13.89	0.64
	Our2	2.77	4.42	1.28	13.54	0.66

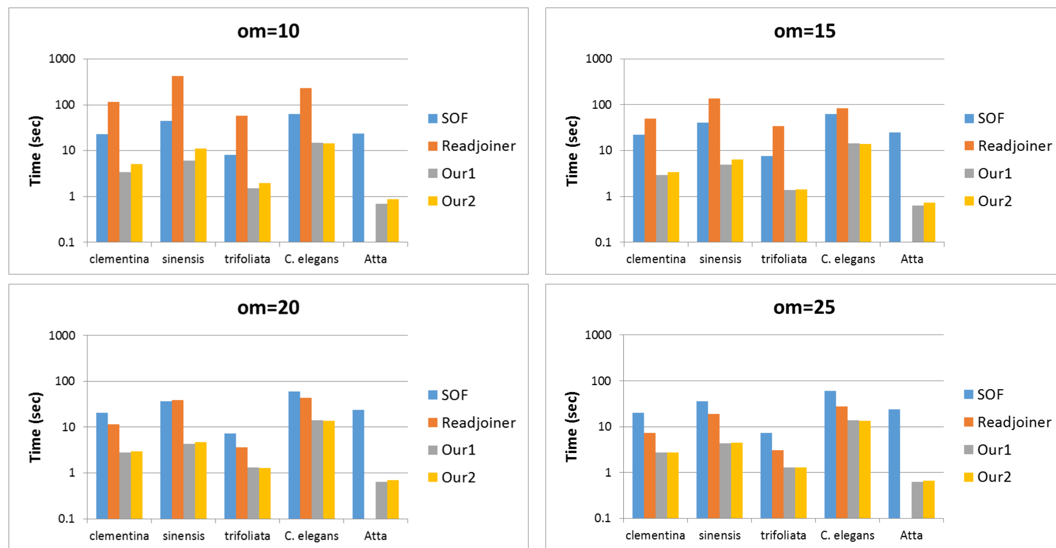


Figure 2 Running time (in second) of algorithms for real datasets (y-axis in log scale).

time for the matching phase, and the time to write the output (For Readjoiner, it is the overlap phase of the whole sequence assembly). The labels of Our1 and Our2 mean our algorithm with option $output = 2$ and $output = 3$, respectively. For SOF, option $-o$ 2 is used. Our2, SOF with $-o$ 2 and Readjoiner return the same output, which includes all overlaps for each pair of input strings. Our1 solves the problem of APSP matching as it is defined (i.e., it returns the longest overlap for each pair of input strings). Readjoiner does not have an option to return the longest overlap for each pair of input strings. SOF finds the longest overlap only when it returns the matrix Ov (with option $-o$ 1), but when it returns a list of $(i, j, Ov[i, j])$ as output (with option $-o$ 2), it returns all overlaps for each pair of input strings and there is no way to return the longest overlap for each pair. In our algorithm we set c_1 to 16 and c_2 to 8 in all experiments.

Table 3 Running time (in second) of algorithms for random datasets.

om	algorithm	rnd1	rnd2	om	algorithm	rnd1	rnd2
10	SOF	109.16	258.82	20	SOF	108.40	252.59
	Readjoiner	481.10	1322.40		Readjoiner	31.67	149.51
	Our1	5.26	13.44		Our1	4.85	12.67
	Our2	5.14	14.01		Our2	4.74	12.48
15	SOF	108.92	256.03	25	SOF	107.3	250.25
	Readjoiner	123.60	214.52		Readjoiner	7.34	14.70
	Our1	4.86	13.03		Our1	4.77	12.38
	Our2	4.81	12.84		Our2	4.83	12.16

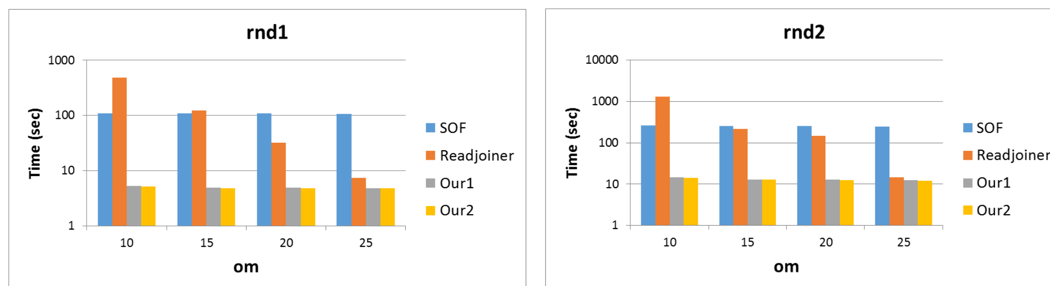


Figure 3 Running time (in second) of algorithms for random datasets (y-axis in log scale).

Table 2 and Figure 2 show the running time (in second) of each algorithm with the real datasets. We carried out experiments when om is 10, 15, 20, and 25. The y-axis (i.e. running time) of Figure 2 is in log scale. Our algorithm outperforms SOF and Readjoiner in all cases. In the experiment with *Atta*, we obtained a huge speed-up compared with SOF because case 1 of the matching step of our algorithm is very effective when m is large. When one of the input strings is very large (its length is over 15 millions in Atta), SOF shows a poor performance. In the experiment with *Atta*, Readjoiner stopped and printed "cannot realloc() memory" for all values of om. Readjoiner is not efficient for small values of om (e.g., $om = 5$). Experimental results show that our algorithm performs well consistently, not depending on one large input string or values of om.

Table 3 and Figure 3 show the running time of each algorithm with random datasets. Again we did experiments when om is 10, 15, 20, and 25. The y-axis (i.e. running time) is in log scale. The performance of our algorithm is better than those of SOF and Readjoiner in all cases. The running time of Readjoiner decreases as om increases.

We computed the average speedup of our algorithm Our2 over SOF and Readjoiner for all experiments. The average speedup of Our2 over SOF for all 28 experiments (5 real datasets + 2 random datasets and 4 values of om) is about 14 and that of Our2 over Readjoiner for all 24 experiments (4 read datasets + 2 random datasets and 4 values of om) is about 18.

Table 4 and Figure 4 show the running time of each algorithm with *parallel* options for real datasets. We used different numbers of threads (1, 2, 4, and 8) and a fixed value 15 of om, which was the value of om in all experiments of the SOF paper [9]. Our algorithm and SOF show reasonable scalability in all experiments. However, SOF does not scale well in the experiment with *Atta* because SOF has poor scalability when one of the input strings is very large. Readjoiner does not show good scalability in all experiments.

The peak value of memory usage is measured by `/usr/bin/time -v`. Table 5 and Figure 5 show the peak memory for SOF, Readjoiner, and Our2 on the real datasets with $om = 15$.

Table 4 Running time (in second) of algorithms with *parallel* options 1, 2, 4, and 8.

algorithm	thread	*clementina*	*sinensis*	*trifoliata*	*C. elegans*	*Atta*
SOF	1	21.98	40.60	7.70	61.94	24.59
	2	11.63	21.41	4.32	34.98	23.23
	4	6.95	11.41	2.26	20.10	23.15
	8	4.61	8.56	1.59	14.22	23.06
Readjoiner	1	50.47	137.54	33.87	84.25	-
	2	42.92	118.02	19.67	80.88	-
	4	41.57	117.39	18.35	77.86	-
	8	43.22	113.64	18.14	71.71	-
Our1	1	2.90	4.90	1.37	14.19	0.64
	2	1.88	3.17	0.99	8.94	0.62
	4	1.28	2.04	0.53	5.79	0.43
	8	1.09	1.97	0.44	4.96	0.42
Our2	1	3.36	6.44	1.42	13.84	0.73
	2	2.34	4.81	0.93	8.44	0.72
	4	1.77	3.52	0.61	5.69	0.54
	8	1.48	3.43	0.50	4.84	0.53

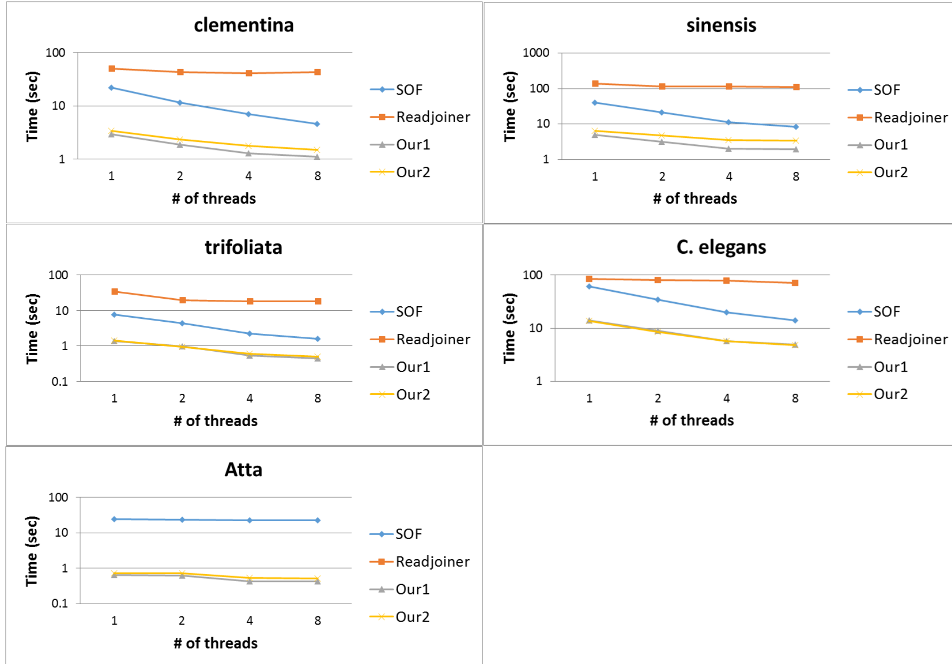


Figure 4 Running time (in second) of algorithms with *parallel* options 1, 2, 4 and 8 (y-axis in log scale).

Table 5 Peak memory usage (in MB) of algorithms.

	clementina	*sinensis*	*trifoliata*	*C. elegans*	*Atta*
SOF	53	81	26	101	111
Readjoiner	191	419	83	88	-
Our2	128	263	89	280	270

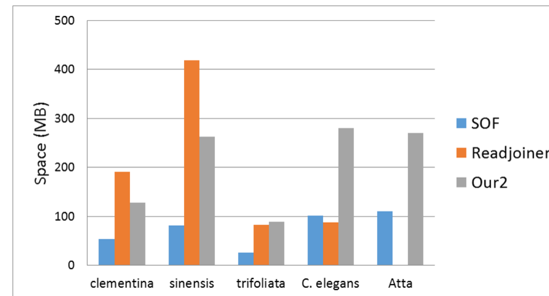


Figure 5 Peak memory usage of algorithms.

Our algorithm uses more memory than SOF in all cases because of additional auxiliary data structures, but the memory usage of our algorithm is still within the optimal bound of $O(N)$ as described in Sections 2.1 and 2.3. Our algorithm uses more memory or less memory than Readjoiner depending on datasets.

4 Conclusion

In this paper we have presented a fast algorithm for all-pairs suffix-prefix matching. The main idea of the algorithm is a combination of a simple but effective data structure for storing input strings and advanced algorithmic techniques for matching to achieve fast running time. Experimental results show that our algorithm runs much faster than previous state-of-the-art algorithms SOF and Readjoiner for APSP matching. Also we obtain reasonable scalability with a parallel implementation of our algorithm.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

2 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

3 Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

4 Maxime Crochemore, Artur Czumaj, Leszek Gąsieniec, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Fast practical multi-pattern matching. *Information Processing Letters*, 71(3-4):107–113, 1999.

5 Hieu Dinh and Sanguthevar Rajasekaran. A memory-efficient data structure representing exact-match overlap graphs with application for next-generation dna assembly. *Bioinformatics*, 27(14):1901–1907, 2011.

6 Giorgio Gonnella and Stefan Kurtz. Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC bioinformatics*, 13(1):82, 2012.

7 Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

8 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41(4):181–185, 1992.

9 Maan Haj Rachid and Qutaibah Malluhi. A practical and scalable tool to find overlaps between sequences. *BioMed research international*, 2015, 2015.

10 Maan Haj Rachid, Qutaibah Malluhi, and Mohamed Abouelhoda. Using the sadakane compressed suffix tree to solve the all-pairs suffix-prefix problem. *BioMed research international*, 2014, 2014.

11 David Hernandez, Patrice François, Laurent Farinelli, Magne Østerås, and Jacques Schrenzel. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome research*, 18(5):802–809, 2008.

12 R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

13 Felipe A. Louza, Simon Gog, Leandro Zanotto, Guido Araujo, and Guilherme P. Telles. Parallel computation for the all-pairs suffix-prefix problem. In *International Symposium on String Processing and Information Retrieval*, pages 122–132. Springer, 2016.

14 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

15 Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005.

16 Enno Ohlebusch and Simon Gog. Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Information Processing Letters*, 110(3):123–128, 2010.

17 Maan Haj Rachid, Qutaibah Malluhi, and Mohamed Abouelhoda. A space-efficient solution to find the maximum overlap using a compressed suffix array. In *Biomedical Engineering (MECBME), 2014 Middle East Conference on*, pages 329–333. IEEE, 2014.

18 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

19 William H. A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix–prefix problem. *Journal of Discrete Algorithms*, 37:34–43, 2016.

20 Sun Wu, Udi Manber, et al. A fast algorithm for multi-pattern searching. Technical report, University of Arizona. Department of Computer Science, 1994.

The Quantile Index – Succinct Self-Index for Top-k Document Retrieval

Niklas Baumstark[1], Simon Gog[2], Tobias Heuer[3], and Julian Labeit[4]

1 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`niklas.baumstark@student.kit.edu`

2 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`simon.gog@kit.edu`

3 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`tobias.heuer@student.kit.edu`

4 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`julian.labeit@student.kit.edu`

Abstract

One of the central problems in information retrieval is that of finding the k documents in a large text collection that best match a query given by a user. A recent result of Navarro & Nekrich (SODA 2012) showed that single term and phrase queries of length m can be solved in optimal $O(m + k)$ time using a linear word sized index. While a verbatim implementation of the index would be at least an order of magnitude larger than the original collection, various authors incrementally improved the index to a point where the space requirement is currently within a factor of 1.5 to 2.0 of the text size for standard collections.

In this paper, we propose a new time/space trade-off for different top-k indexes. This is achieved by sampling only a quantile of the postings in the original inverted file or suffix array-based index. For those queries that cannot be answered using the sampled version of the index we show how to compute the query results on the fly efficiently. As an example, we apply our method to the top-k framework by Navarro & Nekrich. Under probabilistic assumptions that hold for most standard texts, and for a standard scoring function called term frequency, our index can be represented with only sublinearly many bits plus the space needed for a compressed suffix array of the text, while maintaining poly-logarithmic query times. We evaluate our solution on real-world datasets and compare its practical space usage and performance against state-of-the-art implementations. Our experiments show that our index compresses below the size of the original text. To our knowledge it is the first suffix array-based text index that is able to break this bound in practice even for non-repetitive collections, while still maintaining reasonable query times of under half a millisecond on average for top-10 queries.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Text Indexing, Succinct Data Structures, Top-k Document Retrieval

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.15

1 Introduction

Given a large collection of text documents, it remains an important problem in computer science to pre-process them in a way that afterwards user queries about the documents in the collection can be answered efficiently. The general setting for *top-k retrieval* that we concern ourselves with is the following: Given a *collection* of documents $\mathcal{D} = \{d_1, d_2, \ldots, d_N\}$ where each document d_i is a string over some alphabet Σ, a *pattern* string $p \in \Sigma^*$, a *scoring function* $\omega : \mathcal{D} \times \Sigma^* \to \mathbb{R}$ and a parameter $k \in \{1, 2, \ldots N\}$, find the k documents d_i that

© Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit;
licensed under Creative Commons License CC-BY

16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 15; pp. 15:1–15:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

contain p and maximize $\omega(d_i, p)$. If $\mathcal{D}$ and ω are given upfront, we might want to build an *index* data structure that is independent of the pattern p and allows us to process a series of patterns much more efficiently, without having to scan the entire document collection each time. This is surely not feasible for every possible scoring function ω, hence we restrict ourselves to an interesting subset of scoring functions. An example of such a function is *term frequency*, which simply counts how often the pattern appears in the document.

More complex scoring functions, such as Okapi BM25, incorporate more features – namely document frequency and document length – and often lead to better results in practice. In standard information retrieval nomenclature, the type of query described above is called a *phrase query*. Depending on the data structures used, phrase queries can be much harder to solve than *single term queries*, which consist of a single token (such as a word in a natural text). If suffix arrays are used as the basic building block of an index, these two types of queries can be treated the same because no parsing of the text into tokens is required.

There are two main approaches for solving the top-k retrieval problem. *Inverted indexes* are used almost exclusively in practice by real-world search engines such as Apache Lucene [20]. Inverted indexes store all occurrences (or *postings*) of a term in a document in *posting lists*. Decades of research have yielded very compact and fast indexes using this technique. Nevertheless, there are some inherent limitations of the inverted index approach: In order to support phrase queries efficiently, additional information needs to be stored alongside the postings, and heuristics are used to ensure that tokens appear together and in the correct order in the resulting documents.

In 2010, Culpepper et al. [3] presented the first implementation of Hon et al.'s top-k index for phrase queries that is based on *suffix arrays* [9]. Since then, various authors have improved the space efficiency of suffix array-based indexes significantly. The current state of the art can produce indexes that are within a factor of 1.5 to 2 of the text size, while supporting arbitrary phrase queries efficiently [11]. However, space usage is still a clear advantage of inverted indexes, which are typically much smaller than the original text.

We contribute a new technique that can be applied to both inverted indexes and suffix array-based indexes to decrease space usage by trading it for query time. We achieve this by storing only a fraction of the posting lists, and provide an algorithm that can reconstruct the rest of the result list on the fly, in the case where this is necessary because many results are requested – i.e. k is sufficiently large. We implemented our idea on top of an existing top-k index that is based on the framework by Navarro & Nekrich [16]. In particular, we took the implementation from [11] and improved its space usage by a factor of 2 to 2.5 using our quantile sampling method. Our experimental evaluation shows that query times are dependent on the specific query, but are within a factor of 10-20 of the unmodified index in the most unfavourable cases and comparable on average.

2 Related Work

One of the basic tasks closely related to top-k retrieval is *document listing* – enumerating all documents in which a pattern occurs. In 1998, Matias et al. augmented suffix trees in order to solve the document listing problem efficiently [12]. Muthukrishnan introduced an optimal time index for document listing in 2002 [13]. The solution uses a range minimum query data structure to enumerate all distinct document IDs in a lexicographic range. Sadakane proposed a succinct version of the index [18]. As the underlying data structure Sadakane uses a *compressed suffix array* (CSA). For a comprehensive survey on CSAs and basic succinct data structures used in text indexes we refer to [15]. In total Sadakane's solution needs

$|CSA| + 4n + o(n)$ bits to perform document listing, where $|CSA|$ is the size of a compressed suffix array. Since then numerous indexes based on CSAs have been proposed supporting all kinds of different operations. In the following we will focus on the top-k retrieval problem.

In 2009, Hon et al. introduced a general top-k retrieval framework and showed how to solve the problem using an index taking up only a linear number of words [9]. In their work they introduce the notion of weighted arrows in the generalized suffix tree. Queries are answered by enumerating the heaviest arrows pointing out of subtrees of the tree. Subsequently, Navarro & Nekrich showed how to further improve this framework to achieve optimal $O(m+k)$ query time, where m is the pattern length, and reduced its space consumption [16]. This is done by representing arrows as 2-d weighted grid points and using a three-sided prioritized orthogonal range search data structure to answer top-k queries. In 2013, Tsur published the first top-k document framework using optimal $|CSA| + o(n)$ bits of space with poly-logarithmic query times [21]. The query time was reduced by Thankachan & Navarro to $O(k \log^2 k \log^\epsilon n)$ suffix array accesses for any constant $\epsilon > 0$ [17]. For a comprehensive overview of the different problems and solutions concerning document retrieval please refer to the survey put together by Navarro [14].

In recent years there have been numerous practical implementations of suffix array-based document retrieval frameworks. In 2013, Konow & Navarro showed that with a frequency sampling of the suffix tree nodes they can actually implement the Navarro–Nekrich scheme using 3 to 4 times the original text size for non-repetitive collections while achieving top-k query times well below a millisecond [10]. In 2014, Brisaboa et al. introduced the K^2-treap to solve weighted top-k range queries faster and more space-efficiently in practice [2]. In the following year, Gog & Navarro simplified the implementation by Konow & Navarro by introducing a new mapping of suffix array ranges to coordinate ranges of the grid [8]. Their implementation uses 2.5 to 3 times the input size while maintaining comparable query times. Recently, Labeit & Gog proposed a technique to encode the document IDs of the grid points more compactly [11]. With this technique they achieve index sizes of 1.5 to 2 times the input size at the cost of higher query times. [7] gives a comprehensive overview of the techniques developed in this line of research.

Additionally, some of the prior art specializes on repetitive string collections. Gagie et al. show how document listing, top-k retrieval and document counting can be solved while exploiting the properties of repetitive collections [4]. They introduce the *interleaved LCP* array and *precomputed document lists*. Both concepts might also be applicable to top-k retrieval on non-repetitive collections.

3 Quantile Filtering

3.1 Preliminaries

For a collection $\mathcal{D}$ and scoring function ω we define $\mathcal{R}_p = (d_1, d_2, \ldots)$ to be the result list containing all documents in which p occurs in descending score order. In the literature $\mathcal{R}_p$ are often referred to as impact-ordered posting lists of term p. Consider the example collection $\mathcal{D} = \{\texttt{d}_1 : \texttt{ATATT}, \texttt{d}_2 : \texttt{TTATA}, \texttt{d}_3 : \texttt{AATT}, \texttt{d}_4 : \texttt{TTA}\}$, pattern $p = \texttt{TA}$ and term frequency as the scoring function. Then $\mathcal{R}_{\texttt{TA}} = (d_2, d_1, d_4)$ as TA appears twice in d_2, once in d_1 and once in d_4. The top-k retrieval problem can be defined as computing the first k entries of $\mathcal{R}_p$, which we denote by the function $\texttt{top-}k(p, \mathcal{R})$.

Additionally we define $\mathcal{O}cc_p = ((d_1, pos_1), (d_2, pos_2), \ldots)$ to be the list of all occurrences of the pattern p. For our example collection $\mathcal{O}cc_{\texttt{TA}} = ((d_1, 1), (d_2, 1), (d_2, 3), (d_4, 1))$. If we concatenate all documents to one string with a separator character we obtain $\mathcal{D}^* =$

Algorithm 1: The generic query algorithm.

```
Input: Search pattern p, integer k > 0
Output: Top-k documents containing pattern p, sorted by scoring function
1 if k · q ≤ count(p, D*) then
2   return top-k(p, R^q)
3 else
4   Occ_p ← locate(p, D*)
5   return top-k-on-the-fly(Occ_p)
```

ATATT#TTATA#AATT#TTA and can then compute $|\mathcal{O}cc_p|$ by counting the occurrences of p in $\mathcal{D}^*$. We can generate $\mathcal{O}cc_p$ by first computing all locations of p in $\mathcal{D}^*$ and then mapping the locations to pairs of document ID and location within the document. This mapping can be performed using a compressed bit vector. We call a pattern p *right-maximal* if it has two occurrences in $\mathcal{D}^*$ succeeded by different characters. The inner nodes of the suffix tree of $\mathcal{D}^*$ corresponds to all the right-maximal patterns. Finally, we denote the set of all $\mathcal{R}_p$ for all right-maximal patterns as $\mathcal{R}$.

3.2 Basic Framework

In the following we propose a sampling technique of the sorted posting lists $\mathcal{R}_p$. The sampling allows us to use a standard pattern matching index, such as a compressed suffix array (CSA), to reduce the number of postings which need to be represented by our top-k retrieval index. The pattern matching index only needs to support the basic operations locate(p) and count(p), which compute the set $\mathcal{O}cc_p$ and its cardinality, respectively.

Let $q \in \mathbb{N}$ be the *quantile parameter*. Then $\mathcal{R}_p^q$ is defined as the list containing the upper q-th quantile of $\mathcal{R}_p$, i.e. the $\lfloor \frac{|occ_p|}{q} \rfloor$ highest ranked elements of $\mathcal{R}_p$. We denote the set of all $\mathcal{R}_p^q$ lists for all right-maximal substrings p of $\mathcal{D}^*$ as $\mathcal{R}^q$. Our top-k retrieval index represents only the $\mathcal{R}^q$ lists, instead of the $\mathcal{R}$ lists. When solving top-k queries we first compute $|\mathcal{O}cc_p|$ using a pattern matching index. If $k \cdot q \leq |\mathcal{O}cc_p|$ the query can be answered using the top-k index representing $\mathcal{R}^q$. Otherwise the query is answered by computing the result list from $\mathcal{O}cc_p$ on the fly. Note that in the latter case the cardinality of $\mathcal{O}cc_p$ is bounded by $q \cdot k$, which ensures the efficiency of the framework. Algorithm 1 gives the pseudocode for the query algorithm.

With the proposed framework we can solve regular top-k queries on $\mathcal{R}$ using a top-k query on $\mathcal{R}^q$ plus additional operations on a pattern matching index and some on-the-fly computations. We will analyze the asymptotic behaviour of our index under the assumption that the input texts are randomly drawn from a source satisfying the Szpankowski A2 model [19]. This is reasonable to assume for most texts occurring in practice. The A2 model was first used in the context of document listing by Gagie et al. [5]. We use it to bound the height of the suffix tree with high probability by $\Theta(\log n)$.

By choosing different values for q we get different time/space trade-offs between the size of $\mathcal{R}^q$ and the time needed for locate and the on-the-fly computation. The following theorem characterizes the relation between q and the size of $\mathcal{R}^q$.

▶ **Theorem 1.** *For a document collection generated from a source satisfying the Szpankowski A2 model,* $|\mathcal{R}^q| = O(\frac{n \log n}{q})$ *holds with high probability, where* $|\mathcal{R}^q| = \sum_p(|\mathcal{R}_p^q|)$ *for all right maximal substrings* p *of* $\mathcal{D}^*$.

Proof. Each right-maximal substring p of $\mathcal{D}^*$ can be identified by a unique node v in the suffix tree of $\mathcal{D}^*$. We can bound the number of distinct documents in which p occurs by the number of nodes in the subtree rooted at v. Thus the total result list size of p can be bounded by $|\mathcal{R}_p| \leq |subtree(v)|$. Under Szpankowski's A2 model the generalized suffix tree has height $h = \Theta(\log n)$ with high probability (w.h.p). Hence the sum over all such subtrees is bounded by $\sum_v |subtree(v)| \in O(n \log n)$ and consequently $|\mathcal{R}| \in O(n \log n)$ w.h.p. By construction $\mathcal{R}^q$ has a factor of q less elements than the lists in $\mathcal{R}$ so in total we get $|\mathcal{R}^q| = O(\frac{n \log n}{q})$ w.h.p. ◀

3.3 Succinct Self-Index

We apply our method to a state-of-the-art implementation of the Navarro–Nekrich scheme for top-k retrieval. Our choice is based on the fact that their framework already includes a CSA for pattern matching and that recent implementations only need $O(\log \log n)$ index bits per input character[1] to represent $\mathcal{R}^q$. We first give a short overview of the Navarro–Nekrich scheme and then we apply Theorem 1 to show that our index indeed is succinct. For a more in-depth discussion of the ideas underlying the Navarro–Nekrich scheme and the various improvements since its inception please refer to [7].

The foundation of the top-k retrieval framework was laid out by Hon et al. [9]. The key idea is to answer top-k queries by inserting the edges of the suffix trees of the individual input documents into the suffix tree of $\mathcal{D}^*$. The edges are associated with the corresponding document ID and are directed towards the root. Additionally the score $\omega(d, p)$ is computed, where d is the corresponding document ID and p is the pattern represented by the node that the arrow points to. This score is used as the weight of the arrow. Hon et al. observed that top-k queries for a pattern p can be solved by enumerating the k heaviest arrows originating inside and pointing outside of the subtree representing p. Navarro & Nekrich showed that each arrow can be represented as a weighted 2-d grid point [16]. The x coordinates of a grid point is its position in a certain in-order traversal of the arrows by their origin. The y coordinate of a grid point is the tree depth of the tree node that it points to.

The set of all such grid points is called G. Top-k queries are solved by answering three-sided prioritized orthogonal range queries on G. The x range of the query is chosen so that it contains all arrows starting in the subtree corresponding to the query pattern p. The y range is chosen so that only arrows are reported that point out of the subtree corresponding to the query pattern p. Additionally, to increase the practical performance of the framework, *singletons* are handled in a separate document listing data structure. Singletons are arrows that start at suffix tree leaves and thus account for at least half of all the arrows.

▶ **Theorem 2.** *For a document collection generated from a source satisfying the Szpankowski A2 model, applying quantile filtering to the Navarro–Nekrich scheme yields a succinct top-k self-index with high probability. More precisely we obtain an index using $|CSA| + o(n)$ bits of space and $O(k \cdot polylog(n, m))$ time to answer a top-k query for a pattern of length m. Here $|CSA|$ is the space used by the compressed suffix array, which is assumed to support standard operations in time $O(polylog(n, m))$.*

Proof. We choose $q \in \Theta(\log n (\log \log n)^2)$ so that using Theorem 1 we get $|\mathcal{R}^q| \in o(n/\log \log n)$. We only keep those grid points from G that represent results in $\mathcal{R}^q$. We call

[1] To achieve $O(\log \log n)$ index bits per character the input collection needs to satisfy the Szpankowski A2 model and the values of the scoring function have to be encodable in $O(n \log \log n)$ bits.

this the quantile-filtered grid G_q which consequently contains at most $o(n/\log\log n)$ grid points. To store G_q in an augmented wavelet tree we need $O(\log\log n)$ bits per grid point. Thus the total space to store G_q is $o(n)$ bits. We apply coordinate compression on the x range of the grid G_q. Then the mapping of suffix tree leaves to x coordinates can be stored explicitly using $o(n)$ bits. Finally, we use the data structure from [11] to store the document IDs. Using Elias-Fano encoding and a similar analysis as in [11] the IDs can be represented indirectly in $O(\log\log n)$ bits per grid point. So all the document IDs can be represented in $o(n)$ bits. The overall index needs $|CSA| + o(n)$ bits of space with high probability.

The query time is dominated in the worst case by the $k \cdot q$ accesses to the CSA. They are needed to compute $\mathcal{O}cc_p$ in line 4 of Algorith 1. Depending on T_{CSA}, the access time of the CSA, we get an overall query complexity of $O(T_{CSA}(n,m) \cdot k \cdot \log n(\log\log n)^2)$. ◀

This succinct self-index for top-k document retrieval index is called the *quantile index*. For other grid representations (or scoring functions) with higher space usage we can simply adapt q accordingly. It can always be chosen such that the index space is bounded by $|CSA| + o(n)$ bits, potentially at the cost of a higher worst-case query time. In practice q can be chosen empirically such that the grid component of the index requires little space compared to the CSA component.

4 Index Details

Our practical implementation of the quantile index consist of five basic components, listed below. The list does not include the suffix tree of the collection $\mathcal{D}^*$ because it is needed only temporarily and can be discarded after the construction is finished.

- G_q, the 2-d weighted range search data structure over the filtered Navarro–Nekrich grid points. Each grid point corresponds to exactly one arrow and has an additional coordinate x representing a specific order of the arrows which is described in Section 4.1.
- a compressed suffix array (CSA) of the concatenation of all documents in the input collection. It is used to determine the lexicographical suffix array interval $[l,r]$ for a given query, and to implement the fallback case where $q \cdot k > r - l + 1$ and thus we cannot consult G_q. Different sampling rates s lead to different time/space trade-offs for this component.
- a compressed bit vector B marking the positions in the concatenated input $\mathcal{D}^*$ where a new document begins.
- a data structure DOC which stores the document IDs for each point in G_q. In Section 4.3 we describe how to incorporate the ideas from [11] to decrease the space usage of this component. We have implemented variants of our index with and without this optimization.
- a mapping bit vector H_q, which is used to map a query suffix array interval $[l,r]$ to a range of grid coordinates $[x_l, x_r]$.

To clarify how the quantile index works, Algorithm 2 shows how a query pattern p is processed, using term frequency as an example scoring function. In the fallback case, a linear-time selection algorithm such as the one of Blum et al. [1] is used to select the top-k results, after computing the full list of candidate results by scanning the compressed suffix array. Note that our index can support all the scoring functions that the original Navarro–Nekrich framework supports.

Algorithm 2: The query algorithm for the quantile index.

```
   Input: Search pattern p, integer k > 0
   Output: Top-k documents containing pattern p, sorted by term frequency
1  [l, r] ← search(CSA, p)
2  if k · q ≤ r − l + 1 then
3      [x_l, x_r] ← mapSAInterval(H, l, r)
4      points ← topKRangeSearch(G_q, k, [x_l, x_r] × [0, |p|))
5      result ← empty list
6      for p = (x_p, y_p) ∈ points do
7          push(result, DOC [x_p])
8  else
9      freq ← new hash map with default value 0
10     for i = s to e do
11         docid ← rank(B, CSA[i])
12         freq[docid]++
13     result ← partialSort(freq, k)  // select and sort only the k elements with biggest weight
14 output result
```

4.1 Singletons and the Vector H

Each arrow a in the original Navarro–Nekrich framework is associated with a node v_a in the suffix tree of $\mathcal{D}^*$. Exactly one arrow is associated with each leaf of the suffix tree. Such arrows are called singletons because they correspond to postings with frequency one. The top-k index by Konow & Navarro handles singletons using a separate 1-dimensional range minimum data structure for space reasons [10]. One can view our approach as a generalization of this idea: Since for a subtree S of the suffix we only store the most heavy $\lfloor |S|/q \rfloor$ arrows pointing out of it, we only keep arrows that point out of some subtree of size $|S| \geq q$. This is unlikely to be the case for singleton arrows, hence most of them will be eliminated in practice. This allows us to handle singletons the same way as other arrows, without implementing a special case.

To transform the arrows into grid points and apply the Navarro–Nekrich technique, we order them using a specific depth-first traversal of the suffix tree and use the index of an arrow in that sequence as an additional coordinate for the arrow. Algorithm 3 illustrates the traversal. We obtain a sequence $(a_0, a_1, \ldots a_{M-1})$ of arrows, each represented by a tuple $a_i = (y_i, d_i, w_i)$ where y_i is the target depth of the arrow, d_i is the document associated with it and w_i is its weight. We define $x_i = i$ as the x coordinate of the arrow. The tuple (x_i, y_i, w_i, d_i) is a grid point.

Now we compute a bit vector H (ordered by x coordinate), which marks singleton grid points with a one and all other grid points with a zero. We observe that the one bits in this vector correspond exactly to the leaves in the suffix tree and hence to the entries of the suffix array. We can use a rank data structure on top of H (which computes the number of ones in any given prefix of H) to implement the `mapSAInterval` function used in Algorithm 2.

4.2 Construction

Our implementation of the quantile filtering works as follows: The suffix tree is traversed in depth-first order. B-trees are used to represent for each node v the set of grid points that correspond to arrows pointing out of the subtree rooted at v, ordered by decreasing weight. We can compute this set for a node v by merging the sets of its children. If we reuse the

Algorithm 3: Depth-first in-order traversal of the Navarro–Nekrich arrows.

```
1 Function OrderArrows(v)
      Input: Suffix tree rooted at v.
      Output: An ordered list of arrows originating in the subtree v
2     if v is not a leaf then
3         l ← leftmost child of v
4         OrderArrows(l)
5     output arrows originating at v
6     if v is not a leaf then
7         for children r of v except l do
8             OrderArrows(r)
```

Table 1 Collection statistics: number of characters n, number of documents N, average document length, alphabet size σ, and total size in MiB assuming one byte per character.

Collection	n	N	n/N	σ	$\lvert\mathcal{D}^*\rvert$ in MiB
ENWIKIBIG	8,945,231,276	3,903,703	2,291	211	8,535
ENWIKISML	68,210,334	4,390	15,538	206	65
REVISIONS	419,437,305	20,433	20,527	240	400
SOURCES	244,587,464	29,993	8,154	232	233

B-tree of the largest child subtree and merge the others into it, we only need to perform $O(n \log n)$ B-tree insertions in total. For each node v, we enumerate and mark the top q^{-1} fraction of grid points while eliminating points with $y \geq \mathit{depth}(v)$. The total runtime of this traversal is $O(n \log^2 n)$, under the assumption from above that the suffix tree height is bounded by $O(\log n)$ with high probability.

The result of this construction step is a bit vector QFILTER of size M that has a one bit set for all the grid points that appear in at least one top quantile. We can now build the 2-d weighted range search data structure G_q over only the marked grid points. K^2-treaps form a good compromise between space usage and practical performance, even though they do not provide meaningful worst-case guarantees [2]. We store QFILTER and a compressed version of H. Both are used in conjunction to map a suffix array interval to an interval of x coordinates during query time.

4.3 Using Offset Encoding to Compress DOC

The DOC component, which stores the document IDs for all grid points in G_q can be further compressed using the offset encoding technique from [11]. The resulting data structure is a bit vector which can be used in conjunction with the suffix array CSA and bit vector B to decode the document ID for a given grid point. This query time penalty, since suffix array accesses are slower than most other basic operations used in our query algorithm. Section 5 contains a detailed analysis of the compromise between space usage and query time using this approach.

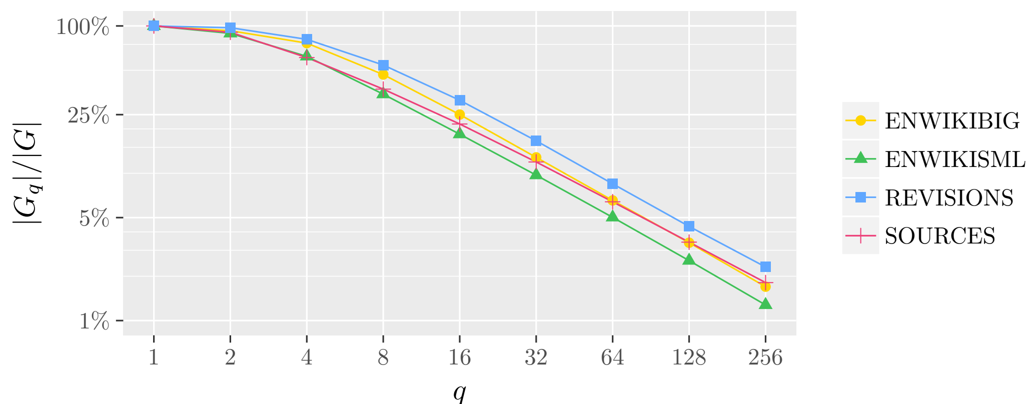


Figure 1 The number of grid points after quantile filtering with varying parameter q, relative to the total number of grid points in the Navarro–Nekrich implementation.

5 Experiments

To thoroughly evaluate the performance of our new top-k index and to compare it against the state of the art, we ran experiments using different input collections. The experiments were executed on a single machine equipped with four Intel Xeon E5-4640 processors, with a combined number of 32 cores and 64 hyper-threads. All experiments were executed in a single thread. The main memory is organized in four banks of 128 GiB each. All programs were compiled using GCC 5.2.0 with optimizations turned on. We used a variety of text collections for our experiments, including two dumps of the English Wikipedia of different sizes (ENWIKISML and ENWIKIBIG), all the revisions of 100 Finnish Wikipedia articles (one revision per document, REVISIONS) and a concatenation of files from the Linux and GCC source tree (SOURCES). Table 1 gives an overview of the basic statistics for each collection. We are comparing four different implementations: NN, the implementation of the Navarro–Nekrich framework by Gog & Navarro [8]; NNL, which adds the offset encoding technique from [11] to NN; Q, our implementation of the quantile index; QL, our quantile index with offset encoding.

For all implementations, the 2-d grid is represented by a K^2-treap as previous work has shown that K^2-treaps use less space than wavelet trees and provide similar query times in practice [7]. We were not able to include an implementation of [4] for time reasons. Their index is designed to perform very well on repetitive collections like REVISIONS, but for ENWIKIBIG its index size and query times are comparable to the NN implementation which is called SURF in their experiments.

The source code used for our experiments can be found at `https://github.com/kit-susi/quantile-index` and includes a script to download the input collections. The basic data structures used by our code are included from in the Succinct Data Structure Library [6].

5.1 Choosing the Index Parameters

Figure 1 shows how the number of stored grid points changes for different q. The experiment indicates that by using $q = 64$ less than ten percent of the grid points need to be stored for all the tested collections. Additionally we can observe that on these test collections the number of grid points $|G_q|$ is actually smaller than suggested by the bound $O(n \cdot \frac{\log n}{q})$. Even for $q \leq \log n$ we see substantial improvements.

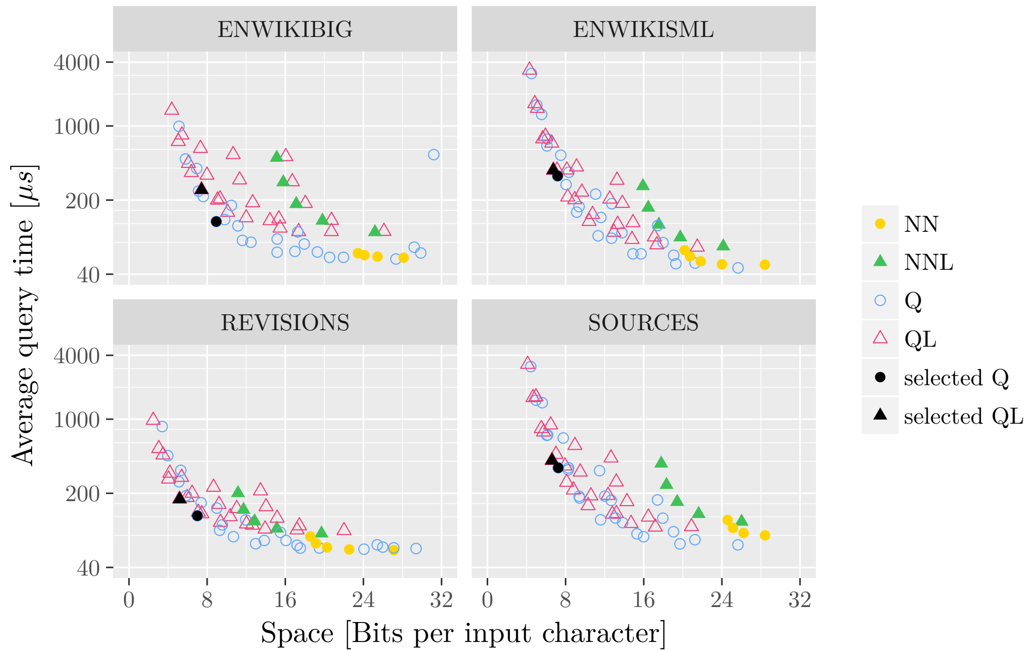


Figure 2 Different time/space tradeoffs resulting from varying quantile parameter $q \in \{8, 16, 32, 64, 128\}$ and CSA sampling parameter $s \in \{4, 8, 16, 32, 64\}$. The x-axis shows the total index size as ratio to the original collection size and the y-axis shows the average top-10 query times for random sampled queries of length $m = 5$ in microseconds. We additional mark the indexes with $(s, q) = (16, 64)$ as they were selected for the other experiments.

Figure 2 shows the different time/space trade-offs in our implementation of the index, resulting from varying CSA sampling parameter $s \in \{4, 8, 16, 32, 64\}$ and quantile parameter $q \in \{8, 16, 32, 64, 128\}$. For further experiments we use $s = 16$ and $q = 64$ as it presents an appealing compromise between query time and index size. Figure 3 shows the total size of the different index data structures with these parameters and highlights how much space is needed by the individual components. As an example, for ENWIKIBIG, our largest collection, the size of the quantile indexes Q and QL is about 65 and 50 percent smaller than the size of NN and NNL indexes, respectively. We note that the offset encoding technique from [11] is not as effective for the quantile index as it is for the NN implementation because the number of grid points is much smaller to begin with in relation to the size of the whole index.

5.2 Query Times

To get a first impression of the query performance of our index, we generated 100,000 random queries by uniformly drawing 5-grams from our biggest input collection. We ran the query set against our implementations, using different values of k, while measuring the individual query times. The results are shown in Figure 4: The average query times per reported document range from 3 to 50 microseconds across all the different implementations and for the document collection ENWIKIBIG. As expected from the results of [11], the variants using offset encoding (NNL and QL) have about a factor of 2 to 3 slower median query times than their counterparts. The plot also clearly shows a large variance in query time for the quantile indexes Q and QL. This is due to the two completely different code paths taken by Algorithm 2 depending on the size of the suffix array (SA) interval corresponding to the query pattern.

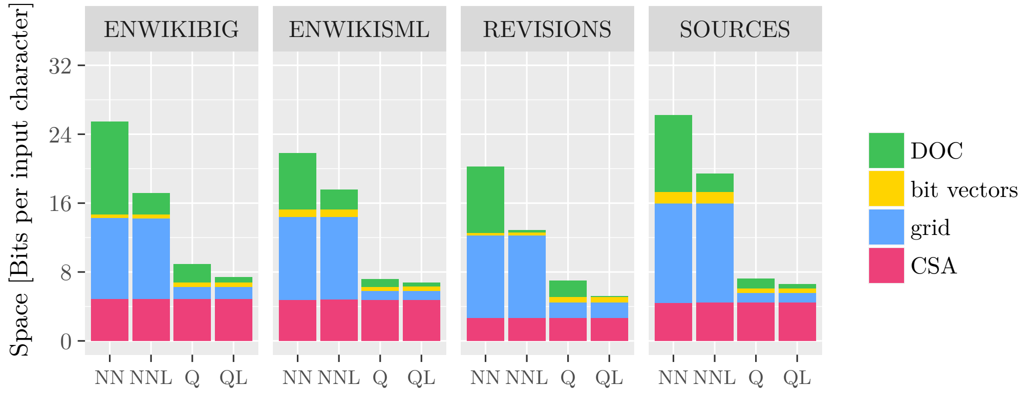


Figure 3 Total index size and size of individual components for $q = 64$ and $s = 16$ in bits per input character. *grid* includes the grid data structure and the RMQ data structure used by NN/NNL to handle singletons postings. *bit vectors* include the H vector and the QFILTER vector in the case of Q/QL.

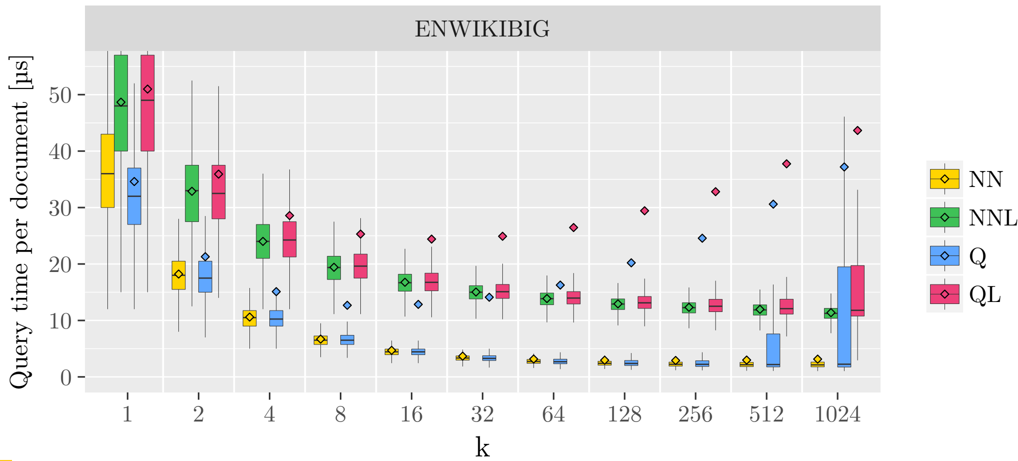


Figure 4 Timings to compute top-k results per reported document, for randomly drawn five-character queries. Both median and mean query times are highlighted, outliers are omitted for readability reasons.

Figure 5 captures the relation between SA interval size and query performance more explicitly. The selection of queries was done in a similar manner to the simple experiment described above, but this time the pattern length was chosen randomly between 3 and 10 for each query so as to vary the sizes of the SA intervals corresponding to the patterns. We requested $k = 10$ results for each invocation of the query algorithm. The plot shows the query times grouped by SA interval size. The separation between the two cases of Algorithm 2 is clearly visible: For the Q and QL implementations, we expect queries with an SA interval of size smaller than $q \cdot k = 640$ to trigger the fallback case where the suffix array range is scanned, which has a running time that is quasi-linear in the SA interval size. Other queries can make use of the more efficient grid search algorithm based on 2-d range queries, which is independent of the SA interval size. This behaviour is clearly exhibited in all four different collections. For ENWIKIBIG, the median query time for the most unfavourable SA interval sizes is still within a factor of 20 of the grid search algorithm in the case of Q. The plot also shows that the offset encoding only affects the grid search case of the query algorithm,

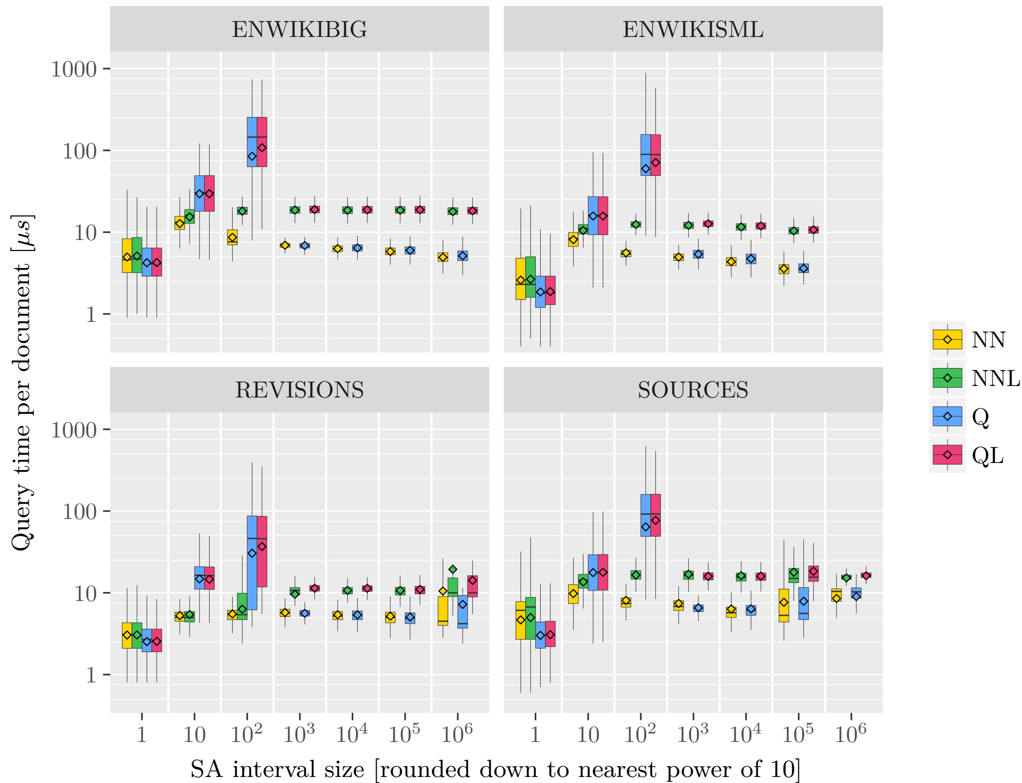


Figure 5 Timings per reported document for $k = 10$, using queries with different suffix array interval sizes. Both median and mean query times are highlighted, outliers are omitted for readability reasons.

not the fallback case. This is because in the fallback case, the suffix array entries for all occurrences of the pattern are known and the B bit vector can be consulted directly to decode the document IDs.

6 Conclusion and Future Work

We have introduced a general technique to reduce the space usage of existing top-k retrieval frameworks. This is achieved by storing only a fraction of the postings in the original index. The sampling is constructed in such a way that the total number of occurrences for query patterns which cannot be answered using the reduced index is bounded by a multiple of k. Hence it is feasible in this case to scan all occurrences and compute the top k results on the fly.

We show an exemplary application of this technique to a state-of-the-art suffix array-based self-index and evaluate its practical performance. Our experiments show that we obtain new time/space trade-offs for the problem at hand. In particular for real-world inputs the compressed suffix array is now the largest component of our proposed index. For non-repetitive texts this is to our knowledge the first index of this kind that is smaller than the original text but can still be used to reconstruct the text fully and support top-k queries with query times typically in the sub-millisecond range. An additional advantage of our framework is that singletons do not need to be handled in a separate data structure and thus it is easier to support more complex scoring functions. In the future we plan to implement

different scoring function used in real-world applications such as Okapi BM25. Currently the implementation of our construction algorithm is suboptimal, making it impractical to evaluate the algorithms on even larger collections. This is partly due to inefficiencies in the implementation and also an incomplete theoretical foundation. For example our construction algorithm works by filtering a pre-computed set of grid points. It should be possible to directly construct the quantile grid from the generalized suffix tree and achieve shorter construction times. Furthermore, the on-the-fly computation of query results for large enough k is currently implemented naively by scanning the suffix array. More elaborate techniques such as the sampled document array could be used to speed up this path of the query algorithm [17].

In this work, we only apply our general framework to one specific implementation. In future work we plan to evaluate whether the framework is also useful when applied to other existing data structures. For example, we are looking to combine a standard inverted index, where postings are ordered by document ID, with multiple small quantile inverted indexes, where postings are ordered by different custom scoring functions. With this construction it may be possible to increase the generality of inverted indexes – which are widely deployed in practice – even further.

References

1. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, pages 448–461, 1973.
2. N. Brisaboa, G. de Bernardo, R. Konow, and G. Navarro. K^2-Treaps: Range Top-k Queries in Compact Space. In *Proc. SPIRE*, pages 215–226, 2014.
3. J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. ESA*, pages 194–205, 2010.
4. T. Gagie, A. Hartikainen, K. Karhu, J. Kärkkäinen, G. Navarro, S. J. Puglisi, and J. Sirén. Document retrieval on repetitive collections. *Information Retrieval*, 2017. To appear.
5. T. Gagie, K. Karhu, G. Navarro, S. J. Puglisi, and J. Sirén. Document listing on repetitive collections. In *Proc. CPM*, pages 107–119, 2013.
6. S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337, 2014.
7. S. Gog, R. Konow, and G. Navarro. Practical compact indexes for top-k document retrieval. *J. Experimental Alg.*, 22(1):article 1.2, 2017.
8. S. Gog and G. Navarro. Improved single-term top-k document retrieval. In *Proc. ALENEX*, pages 24–32, 2015.
9. W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722, 2009.
10. R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. DCC*, pages 5–17, 2013.
11. J. Labeit and S. Gog. Elias-fano meets single-term top-k document retrieval. In *Proc. ALENEX*, 2017.
12. Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv. Augmenting suffix trees, with applications. In *Proc. ESA*, pages 67–78. Springer, 1998.
13. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666, 2002.
14. G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comp. Surv.*, 46(4):article 52, 2014.
15. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1), 2007. Article 2.

16 G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. SODA*, pages 1066–1078, 2012.

17 G. Navarro and S. Thankachan. Faster top-k document retrieval in optimal space. In *Proc. SPIRE*, pages 255–262, 2013.

18 K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Alg.*, 5(1):12–22, 2007.

19 W. Szpankowski. A generalized suffix tree and its (un) expected asymptotic behaviors. *SIAM Journal on Computing*, 22(6):1176–1198, 1993.

20 The Apache Software Foundation. Apache Lucene. `http://lucene.apache.org/`.

21 D. Tsur. Top-k document retrieval in optimal space. *Information Processing Letters*, 113(12):440–443, 2013.

Online Construction of Wavelet Trees*

Paulo G. S. da Fonseca[1] and Israel B. F. da Silva[2]

1 **Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil**
paguso@cin.ufpe.br

2 **Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil**
ibfs@cin.ufpe.br

Abstract

The wavelet tree (WT) is a flexible and efficient data structure for representing character strings in succinct space, while allowing for fast generalised rank, select and access operations. As such, they play an important role in modern text indexing methods. However, despite their popularity, not many algorithms have been published concerning their construction. In particular, while the WT is capable of representing a sequence of length n over an alphabet of size m in $n \lg m + o(n \lg m)$ bits, much more space is typically used for its construction. Here we propose an $O(n \lg m)$-time *online* method for the construction of the WT, requiring no prior knowledge about the input alphabet. The proposed algorithm is conceptually simpler than other state-of-the-art methods, while having comparable time performance and being more space-efficient in practice, since it performs just one pass over the input text and uses little extra space other than for the structure itself, as shown both theoretically and empirically.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Wavelet tree, Online construction

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.16

1 Introduction

The wavelet tree is a fundamental data structure proposed by Grossi, Gupta and Vitter [10], whose myriad virtues have been widely recognised since its introduction [6, 18, 17]. They are used for representing large character strings in tight space, while allowing for some specific questions about their composition, for instance 'how many occurrences of a particular letter are there between positions 3- to 8-billion?', to be answered very quickly, without having to probe the string. It is thus a cornerstone of modern string matching techniques.

Despite having been introduced for well over a decade, not many papers have been published that deal specifically with their construction. In this paper we present a simple and efficient method for the online construction of wavelet trees. We contend that the algorithm proposed herein is among the most efficient methods in practice, both in time and space, and provide experimental evidence to support this claim.

We begin by describing the data structure, its main operations and space requirements. Then we present our construction algorithm with a theoretical analysis of its costs. After that we discuss previous related work, and where our contribution sits in the context. Finally, we report on our experimental analysis and state our conclusions.

* This work was supported by the brazilian Conselho Nacional de Pesquisa – CNPq (Project MCTI/CNPq/Universal 449842/2014-2) and the Fundação de Amparo à Ciência e Tecnologia de Pernambuco – FACEPE (Project APQ-0587-1.03/14).

© Paulo G. S. da Fonseca and Israel B. F. da Silva;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 16; pp. 16:1–16:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

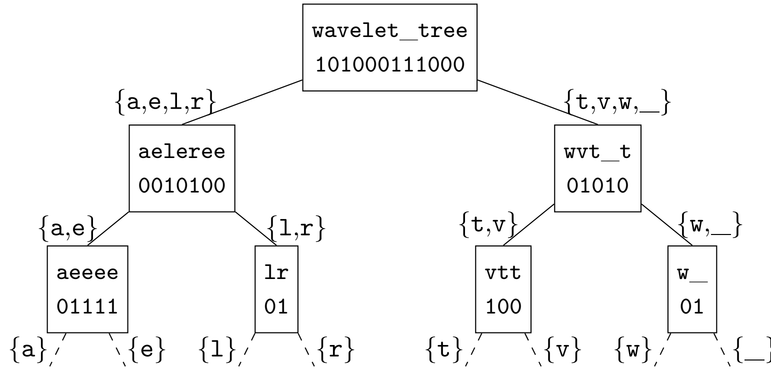


Figure 1 Balanced WT of $T = \texttt{wavelet_tree}$ over $\mathcal{A} = \{\texttt{a}, \texttt{e}, \texttt{l}, \texttt{r}, \texttt{t}, \texttt{v}, \texttt{w}, _\}$. Only the bit vectors are actually stored. The strings and alphabets are shown for illustration purposes only.

2 The Wavelet Tree

We consider strings $T = t_0 \cdots t_{n-1}$ over a finite alphabet $\mathcal{A} = \{a_0, \ldots, a_{m-1}\}$. For any subset $\mathcal{A}' \subseteq \mathcal{A}$, let $\text{proj}(T, \mathcal{A}')$ be the noncontiguous, possibly empty, subsequence of T consisting of all its positions in $\mathcal{A}'$. We call this the *projection* of T on $\mathcal{A}'$. In particular, $\text{proj}(T, [l:r])$, with $0 \leq l, r \leq m$, denotes the subsequence of T made of its characters in the range $\mathcal{A}[l:r] \equiv \{a_l, \ldots, a_{r-1}\}$. For example, if $\mathcal{A} = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$ and $T = \texttt{adbcabdcb}$, then $\text{proj}(T, [1:3]) = \texttt{bcbcb}$. Complementary, we define the *support* of T as $\text{supp}(T) = \cup_{i=0}^{n-1}\{t_i\}$, that is, the subset of $\mathcal{A}$ consisting of the characters in T. Hence we have $\text{proj}(T, \text{supp}(T)) = T$, and $\text{supp}(\text{proj}(T, \mathcal{A}')) \subseteq \mathcal{A}'$.

▶ **Definition 1.** A wavelet tree (WT) of a nonempty string T with support $\mathcal{A}$ is a digital search tree recursively defined as

$$W(T, \mathcal{A}) = \begin{cases} \bot, & \text{if } |\mathcal{A}| = 1, \\ \begin{array}{c} B(T, \mathcal{A}_0, \mathcal{A}_1) \\ \swarrow \quad \searrow \\ W(\text{proj}(T, \mathcal{A}_0), \mathcal{A}_0) \quad W(\text{proj}(T, \mathcal{A}_1), \mathcal{A}_1) \end{array}, & \text{otherwise,} \end{cases} \tag{1}$$

where

- $\mathcal{A} = \mathcal{A}_0 \cup \mathcal{A}_1$ is a nontrivial partition of the alphabet,
- The root consists in an indicator bit array $B(T, \mathcal{A}_0, \mathcal{A}_1) = b_0 \cdots b_{n-1}$ of length $n = |T|$, such that $b_i = 0$, if $t_i \in \mathcal{A}_0$, or $b_i = 1$, if $t_i \in \mathcal{A}_1$,
- The left and right subtrees, $W(\text{proj}(T, \mathcal{A}_0), \mathcal{A}_0)$ and $W(\text{proj}(T, \mathcal{A}_1), \mathcal{A}_1)$ correspond to wavelet trees of the projections of T over the subalphabets $\mathcal{A}_0$ and $\mathcal{A}_1$, respectively, and
- $\bot$ represents a null (empty) tree, which is the base case for unary alphabets.

The most common case of the definition of $W(T, \mathcal{A})$ happens when, at each node, the alphabet is split into two halves, as shown in Figure 1. In this case, the WT is said to be *balanced*.

The WT data structure provides for generalised *access*, *rank*, and *select* queries. Given $W = W(T, \mathcal{A})$, $W.\text{access}(i)$ returns the character t_i, $W.\text{rank}(c, i)$ returns the number of occurrences of character c in T up to (and including) position i, and $W.\text{select}(c, j)$ returns the position of the jth occurrence of character c in T, so that if $i = W.\text{select}(c, j)$, then

$W.\text{rank}(c, i) = j$. In order to do so efficiently both in time and space, the WT employs specialised bit vectors supporting constant-time binary rank and select primitive operations, at the cost of only a sublinear amount of extra bits $E(r) = o(r)$, where r is the lenght of the raw sequence of bits [1, 11]. Hence the balanced WT requires $\approx \lg m(n+o(n)) = n \lg m+o(n \lg m)$ bits of space for the bit vectors since, at every level of the tree, each letter of the text is represented exactly once. On top of that, we have one pointer per node, each taking $O(\lg n)$ bits, thus $O(m \lg n)$ bits for the tree structure. This $O(m \lg n)$ factor can actually be avoided, as discussed in [18], and we can have a representation with just one bitvector and no pointers in $n \lg m + o(n \lg m)$ bits of space, which qualifies the WT as a *succinct* data structure [15], since the uncompressed sequence takes $n \lg m$ bits. Moreover, the generalised access, rank and select queries can be answered in $O(\lg m)$ time by following root-to-leaf paths and performing rank and select queries on their bit arrays.

3 WT construction

The naive recursive procedure for building $W(T, \mathcal{A})$ that mirrors Definition 1 requires partitioning $\mathcal{A}$, computing the indicator bit array of the root node, $B(T, \mathcal{A}_0, \mathcal{A}_1)$, based on each character of T belonging to either $\mathcal{A}_0$ or $\mathcal{A}_1$, and recursively building the left and right subtrees from the projections of T over these subalphabets. In the balanced WT, at each node subalphabets can be represented as $[l : r]$ pairs, and the bit array and the projections can be computed in one pass over the corresponding substring. In total, we have $O(n \lg m)$ time complexity. Nevertheless this strategy requires creating and maintaining several intermediate substrings (the projections) through the recursion stack, and going over multiple copies of the same character of T.

The main motivation of our method is to completely avoid the costly operations of explicitly partitioning the alphabet and projecting the strings, or any other expensive substring manipulation like character counting or sorting. We want to process the input string in one single pass and so our proposed algorithms are *online*, that is we scan each character of T exactly once from left to right, and maintain no extra copies.

For the sake of presentation only, we consider two different scenarios. In the first case, the support alphabet is known in advance. This situation is used as an introduction to the more general case where the alphabet is unknown, and revealed only as the string is scanned.

3.1 WT construction with known alphabet

The first procedure, shown in Algorithm 1, is used to build a WT for a string T whose support alphabet is given. The idea is very simple and consists in, first, initialising an empty WT, called *template*, and then filling the bit arrays in one scan of T. For every scanned character t_i, the algorithm follows the corresponding (unique) root-to-leaf path, appending the necessary bits to the nodes on its way.

There is just one subtlety that will prove important in what follows. We have been accustomed to depictions of balanced WTs where the alphabet is literally cut in half, with the first $\lceil m/2 \rceil$ symbols assigned to the first subalphabet, and the remaining $\lfloor m/2 \rfloor$ to the second one. This is equivalent to assigning a_j to left or right subtrees down the root based on the binary representation of j from left to right, that is, from the most to the least significant bit. However, this need not be the case in general, for it suffices that the partition be evenly sized so as to keep the tree balanced, irrespective of which symbol goes where. So, instead, we choose to follow the bits of j from right (lsb) to left (msb). This results in symbols being assigned to the left/right subalphabets in an alternate fashion.

Algorithm 1 Balanced WT online construction with known alphabet

```
1:  Algorithm WT0 (T = t_0 ··· t_{n−1}, A = {a_0, ..., a_{m−1}})
2:      root ← buildTemplate([0 : m])
3:      for i ← 0, ..., n − 1 do
4:          cur ← root
5:          j ← A.index(t_i)
6:          while cur ≠ ⊥ do
7:              if j mod 2 = 0 then
8:                  cur.B.append(0)
9:                  cur ← cur.left
10:             else
11:                 cur.B.append(1)
12:                 cur ← cur.right
13:             j ← j ≫ 1            ▷ Right shift
14:     return root
```

```
1:  Algorithm buildTemplate ([l : r])
2:      if (r − l) = 1 then
3:          return ⊥
4:      root ← new empty WT node
5:      h ← ⌈(l + r)/2⌉
6:      root.left ← buildTemplate([0 : h])
7:      root.right ← buildTemplate([h : m])
8:      return root
```

▶ **Proposition 2.** *Algorithm 1 builds $W(T, \mathcal{A})$ in $O(n \lg m)$ time, using $\alpha n \lg m + O(1)$ bits of space beyond the size of the WT and $\mathcal{A}$, for some constant $0 < \alpha \leq 1$.*

Proof. The procedure buildTemplate is used to create a strictly binary tree with m terminal null nodes ($\perp$), hence $m - 1$ actual nodes, of which $\lfloor m/2 \rfloor$ are leaves. Since the nodes are empty, just $O(m)$ time is needed.

For each character read, the algorithm visits $\lceil \lg m \rceil$ nodes on the path from the root to a leaf. At each node, a bit is appended to a growing bit vector, which can be done in constant amortised time by using dynamic arrays [5, Sec 17.4]. Therefore we have $O(n \lg m)$ time for filling the previously built template. Since $m = |\text{supp}(T)| \leq |T| = n$, this phase dominates the cost, and we have $O(n \lg m)$ time for the entire construction process.

As for the space, notice that only the currently scanned symbol of T is used in the main loop, and so only that symbol needs to be kept in memory at any given time. Therefore we would only need space for the WT itself (bit arrays, pointers, etc.), plus the alphabet, plus a small constant amount of bits for the working variables. However, the dynamic arrays require extra space to ensure the constant amortised time per append operation. At any time, each level of the tree has as many *used* bits as the number of characters read so far, but at most α times as many bits may be physically allocated, with typical values of α ranging from 1.5 to 2. Hence at most $\alpha n \lg m + O(1)$ bits of total extra space may be required. ◀

▶ Remark. (i) In this analysis, we are not explicitly accounting for the work to build the auxiliary structures of the bit vectors, needed for constant-time rank/select. We can safely assume, however, that this work is linear on the size of the arrays, and thus the proposition remains valid. (ii) We also assume that the alphabet data type supports the computation of the index (rank) of a given character in constant time, which is easily accomplished by many dictionary data structures albeit with different space-time tradeoffs.

3.2 WT construction with unknown alphabet

We now turn to the online construction of the balanced WT of T with no prior knowledge about its support alphabet. Contrary to Algorithm 1, the WT cannot be laid out in advance

in this case. Instead, the alphabet, and consequently the shape of the tree itself, must be updated along with its content as the characters are scanned.

Let us consider first the update in the tree structure as the size m of the alphabet grows. The structure of the WT reflects an hierarchic binary decomposition of the alphabet, with an 1:1 correspondence between nodes and subalphabets. In order to keep the partition balanced as m increases, each *new* symbol will be successively assigned to either the left or right subalphabet in an alternate fashion, as with the previous case.

Now, consider the update from $W^{(i)} = W(T[0:i], \mathcal{A}^{(i)} = \text{supp}(T[0:i]))$ to $W^{(i+1)} = W(T[0:i+1], \mathcal{A}^{(i+1)} = \text{supp}(T[0:i+1]))$ upon reading t_i. If $t_i = a_j$ for some $a_j \in \mathcal{A}^{(i)}$, then the update is similar to one iteration of Algorithm 1, for the symbol is already represented in $W^{(i)}$. If, on the other hand, $t_i \notin \mathcal{A}^{(i)}$, then the update goes as follows. Let $s = |\mathcal{A}^{(i)}|$ be the size of the current alphabet. Starting at the root, if s is even (lsb=0), then the next symbol $a_s = t_i$ should be assigned to the left subalphabet. Thus the next position of the root bit array should be set to 0 and the left subtree should then be updated. If s is odd (lsb=1), we set $root.B[i]$ to 1 and proceed down to the right subtree. Appending a new bit to a bit vector does not affect its previous positions because of the alternating partitioning pattern. By following the same procedure at each node down the path, we may eventually reach an endpoint (a leaf in this case) corresponding to a binary subalphabet, say $\{a_p, a_q\}$ with $p < q < s$. After the addition of the bit concerning $t_i = a_s$ to this endpoint, the subalphabet becomes $\{a_p, a_q, a_s\}$, at which point this node has to be further split into a left child accounting for the binary subalphabet $\{a_p, a_s\}$, and an 'implicit' ($\perp$) right child corresponding to $\{a_q\}$. The new left child bit array should now represent $\text{proj}(T[:i+1], \{a_p, a_s\})$ with $a_p \equiv 0$ and $a_s \equiv 1$, thus being in the form 0^k1, for the only occurrence of a_s corresponds to the newly added t_i. Another possibility is that the endpoint represents a ternary alphabet $\{a_p, a_q, a_r\}$. In this case it would have a left child but not a right child, which is where the update should proceed to. So, a new right child has to be added to represent $\text{proj}(T[:i+1], \{a_q, a_s\})$. An endpoint cannot correspond to a subalphabet of size ≥ 3, or it would have split in previous iterations and the update could have continued to one of its children.

The procedure outlined above is given in Algorithm 2. The WT1(T) algorithm returns a reference to the root of the WT, as well as the support alphabet uncovered during the construction. It uses two main functions to incrementally build the WT. The update($root, \mathcal{A}, c$) function updates the bit vectors of the nodes in the appropriate root-to-leaf path of the current WT, adding information about t_i. As mentioned, this procedure is similar to one iteration of Algorithm 1. It returns a pointer to the endpoint *term* where the update stopped, plus the size *sterm* of the subalphabet represented by that node *after* the update. These values are fed into the testAndSplit procedure, which tests whether *term* needs to be further split and, if so, creates the appropriate child nodes.

▶ **Proposition 3.** *Algorithm 2 builds $W(T, \mathcal{A})$ in $O(n \lg m)$ time, using $\alpha n \lg m + O(1)$ bits of space beyond the size of the constructed WT and $\mathcal{A}$, for some constant $0 < \alpha \leq 1$.*

Proof. As in Algorithm 1, the total work amounts to creating the tree structure and filling the bit arrays. The only difference is that now these steps are interleaved rather than performed one after the other. The work required for creating the tree structure alone is the same in either case, since the resulting trees are isomorphic. As for the operations required for filling the bit arrays, we notice that every bit of the WT is set exactly once by a call to a bit array append operation, and is never modified thereafter. So, each bit is added in $O(1)$

Algorithm 2 Balanced WT online construction with unknown alphabet

```
1: Algorithm WT1 (T = t_0 ··· t_{n-1})
2:     A ← {}
3:     root ← new WT node
4:     for i = 0, . . . , n − 1 do
5:         term, sterm ← update(root, A, t_i)
6:         testAndSplit(term, sterm)
7:         A ← A ∪ {t_i}
8:     return root, A

1: Algorithm testAndSplit(term, sterm)
2:     if sterm ≤ 2 then
3:         return
4:     chd ← new WT node
5:     b ← (sterm − 1)  mod 2
6:     k ← term.B.count(b) − 1
7:     chd.B.append(0^k)
8:     chd.B.append(1)
9:     if sterm = 3 then
10:        term.left ← chd
11:    else if sterm = 4 then
12:        term.right ← chd

1: Algorithm update(root, A, c)
2:     if c ∈ A then
3:         s ← A.index(c)
4:         newc ← 0
5:     else
6:         s ← |A|
7:         newc ← 1
8:     cur ← root
9:     prev, sprev ← ⊥, 0
10:    while cur ≠ ⊥ do
11:        prev ← cur
12:        sprev ← s
13:        if s mod 2 = 0 then
14:            cur.B.append(0)
15:            cur ← cur.left
16:        else
17:            cur.B.append(1)
18:            cur ← cur.right
19:        s ← s ≫ 1
20:    return prev, sprev + newc
```

amortised cost by using dynamic arrays.[1] Hence we have the same $O(n \lg m)$ time for the entire construction procedure.

The space requirements analysis is identical to that of Proposition 2. ◀

4 Related work

As mentioned, the standard recursive procedure for constructing WTs requires $O(n \lg m)$ time. However the constants involved are somewhat large in practice, since it requires manipulating strings at each node. The space requirements are also significant because of the explicit projected copies of T at each level of the recursion, for a total $O(n \lg^2 m)$ bits in the worst case. As pointed out by some authors [19, 3], this space can be reduced to $O(n \lg m)$, on top of the original sequence, by reusing parts of the same allocated space through the recursion. This approach is implemented in LIBCDS [2]. The string manipulations remain costly nonetheless.

One way to reuse the same copy of the input sequence is by sorting their symbols according to the level of the recursion/WT. For instance, in a ‘classic’ WT, all the symbols whose highest bit is 0/1 will be to the left/right of the root. Thus, is we perform a stable, in-place sort of the sequence based on that bit, we will have all its symbols in the correct order for the next level, and the projections can now be represented by $[l : r]$ pairs. On the next level, the sort is based on the second highest bit, and so on. Tischler [25] explores these

[1] Actually, setting the initial bits of a newly created leaf (line 7 of procedure testAndSplit) may be a bit ‘cheaper’ in practice because we amortise the cost of getting to that node and, moreover, appending multiple zeros can be implemented a bit faster.

ideas to give space-efficient constructions of WTs in BFS and DFS order, using between constant and $O(\sqrt{n}(\lg m + \lg n))$ bits of extra space depending on a parameter c that also implies a $\lg c$ runtime multiplier. Claude and coauthors [3] also proposed space-efficient construction algorithms that are however more complex. Their most efficient algorithm runs in $O(n \lg n \lg^2 m + C(n \lg m))$ using $O(\lg m \lg n) + E(n \lg m))$ extra bits, where $C(r)$ and $E(r)$ denote the time to build and the space used by the auxiliary rank/select structures of the bit vectors. Such time and space are explicitly accounted for because the construction process depends on these structures. This algorithm is also destructive, meaning that it overwrites the original input sequence.

Simon Gog maintains SDSL, a very complete and mature C++ library of succinct data structures [9], containing the implementation of several methods described in the literature. This library has a few implementations of WTs with support to various topologies, alphabets, and bit vectors. The standard construction procedure consists in first reading the input to compute individual character counts and the effective support alphabet size, then initializing the tree structure, and finally filling in the node contents, much like in Algorithm 1.[2] Our novelty relative to this algorithm lies in the fact that our Algorithm 2 is online on the input and does not require precomputing the alphabet, which is both more general in theory, and can represent practical advantages, for example, in the context of streaming applications.

Some authors have recently proposed parallel algorithms for the construction of WTs. Fuentes-Sepúlveda and collaborators [7] explore the fact that the node corresponding to a certain character, at any given level of a classic WT, can be accessed by the highest bits of its index in the alphabet (as explained in Section 3.1), to build multiple levels in parallel. This $O(n \lg m)$ work is performed in $O(n)$ depth with $O(\lg m)$ processors, but the algorithm assumes continuous integers alphabet that need to be given as input. Shun [24] follows a similar ideas but builds the WT level-by-level, each level requiring $O(n)$ work in $O(\lg n)$ depth, for a total $O(\lg n \lg m)$ depth. These bounds are further improved by using stable sort algorithms to sort the characters of the input at each level of the tree by its highest bits so as to put them in the right order. However, this significantly increases memory demands.

Very recently, Munro and coauthors [13] presented the first algorithm to build a WT in $O(n\lceil \lg m/\sqrt{\lg n}\rceil)$, therefore a $\sqrt{\lg n}$-speedup over previous methods. Their technique is based on the use of bit-parallelism to pack the information concerning group symbols of the input sequence in words and process them together, thus achieving a time that is actually smaller than the size of the structure. Unfortunately, no implementation is available to the best of our knowledge [12].

5 Experimental analysis

In order to evaluate the applicability of our algorithm, we implemented a prototype in C and tested it with data obtained from the Pizza&Chili corpus website [20], as summarised in Table 1. For each data set, we created input files of sizes 2, 4, 8, 16, 32, 64, and 128 MB, by cropping the original files. These input sizes were shown to be enough for establishing a pattern in our experiments, as seen below. In addition, because the theoretical analysis of the previous section assumes the word-RAM model, these sizes also allow for a more clear comparison, unaffected by virtual secondary memory usage factors. We have then a total of $5 \times 7 = 35$ input sequences of varying length (n) and alphabet size (m), the two main parameters that affect the construction time and memory. We wanted to assess how our

[2] We have not found a description or analysis of this algorithm in the literature.

Table 1 Data sets used in the experiments. Original sources are indicated in [20].

Data set Id	Brief description	Total size	Alphabet size (m)	H_0 Entropy
dna	Gene DNA sequence	386 MB	16	5.465
proteins	Aminoacid sequence	1.2 GB	27	4.206
xml	XML bibliography data	283 MB	97	5.262
sources	C/Java source files	202 MB	230	5.537
english	English language texts	2.1 GB	239	4.525

implementation behaved in practice, relative to the theoretical predictions of Propositions 2 and 3.

Our implementation, herein identified as **fs**, consists in a simple pointer-based WT with $O(\lg m)$-time access, rank, and select operations, using a straightforward implementation of the uncompressed combined sampling rank and select bit vector [21]. We used no external libraries other than the C standard API. It is important to note that, although the alphabet is actually known for each data set in Table 1, this information is not fed into our algorithm. Instead, only the path to the text file is provided, which is read in one single pass as described in Section 3.2. In this case, the use of the C standard I/O (`stdio`) file manipulation functions [14] effectively results in a buffered input stream behaviour.

For comparison sake, we benchmarked our prototype together with other publicly available WT code. We wanted to assess the algorithms in a realistic situation where only the path to the file with the input is given. However, each implementation has different interfaces and so, for some of them, we wrote minimal wrapper code to read the input and pass it to the WT constructor using the appropriate internal data structures. We account for those operations as part of the algorithms to have a more uniform comparison. Here is a summary.

libcds. The LIBCDS [2] is a compressed data structures library whose WT code has been used in other comparative studies, including [3, 7]. The input has to be loaded into internal structures.

sdslil, sdslrrr. We used two variants of the WT construction example provided with the SDSL distribution [9]. The first variant, herein identified as sdslrrr, is identical to the example and uses RRR bit arrays[23], which is a compressed structure. Since our implementation does not use compression, we have also added a version based on uncompressed bit array, herein identified as sdslil. No input preprocessing was necessary.

pwt, dd. We used the code made available by the authors of [7]. The code requires that the text be encoded in a contiguous integer alphabet whose size has to be informed, and so we had to made the appropriate conversions.

swt. We used the code of the `serialWT` version made available by Shun [24], which gave the best results among the provided variants. No input conversion was needed.

When provided, the original scripts were used to build the executables. The only eventual modifications were made to ensure that each implementation was compiled with the same `-O3` optimisation flag. The actual code can be obtained from the address indicated at the end of this paper. All experiments reported here were performed on a portable computer equipped with a 2.0GHz Intel® Core™ i7-3537U CPU, 8 GB of RAM, and running 64-bit (Ubuntu 16.04 LTS) Linux ver. 4.4.0, with GCC ver. 5.4.0.

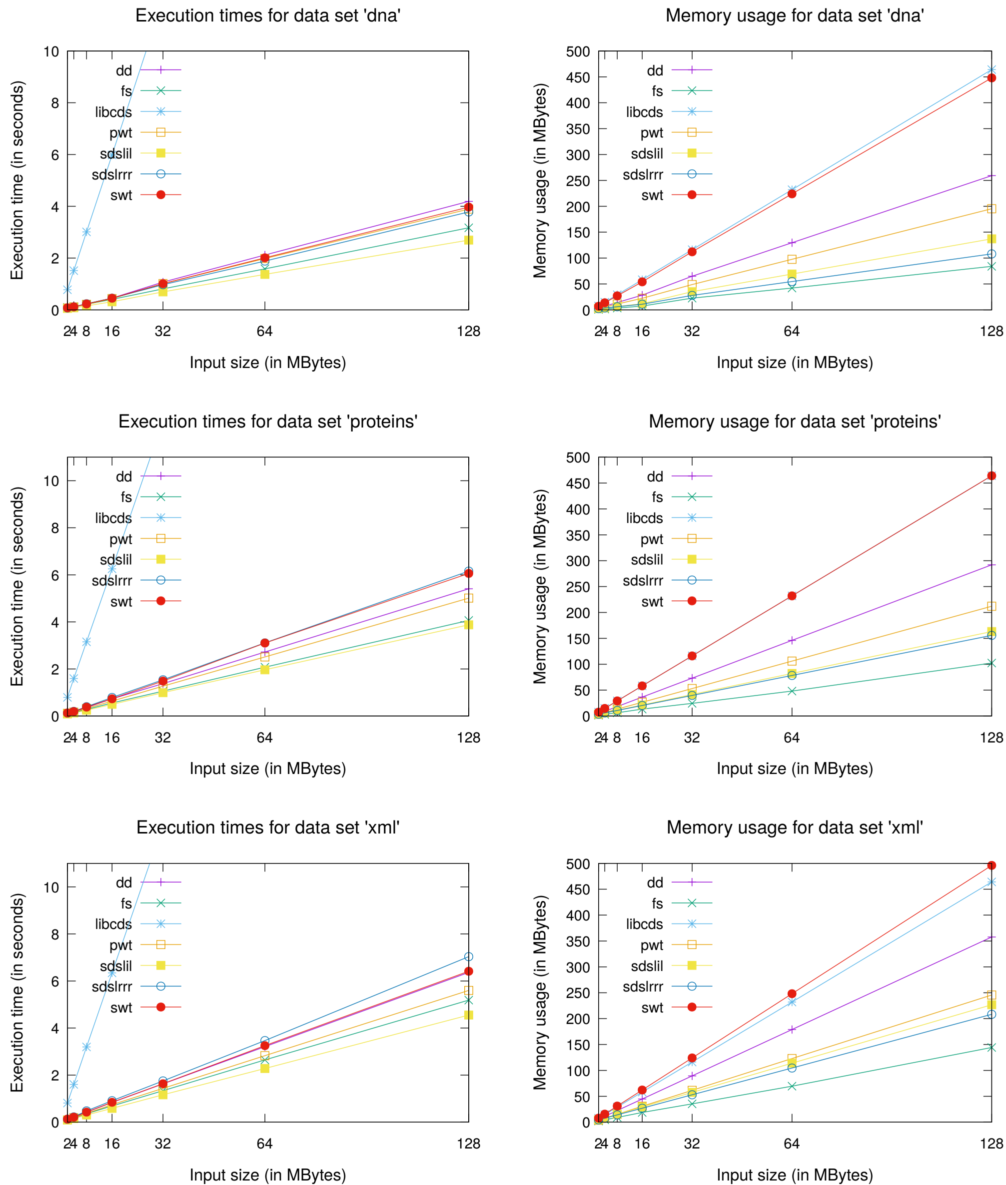


Figure 2 Experimental results (Part 1 – continued on next page). On the left, the average execution times, on the right, the corresponding memory usage information. Alphabet size (m) grows from top to bottom row.

5.1 Time experiments

We used the input files to build WT with all algorithms and measured the *total* execution time using the built-in Bash shell `time` command, including the user time, and the time spent by the system on behalf of the process. The tests were repeated 5 times for each input file and the average results are graphically summarised in the left column of Figure 2.

As it can be seen, a clear linear dependence on the length of the text is shown for all algorithms and for any given alphabet size. The logarithmic dependence on m is also fairly visible from the plots, by noticing, for instance, that de average times for the `dna` data

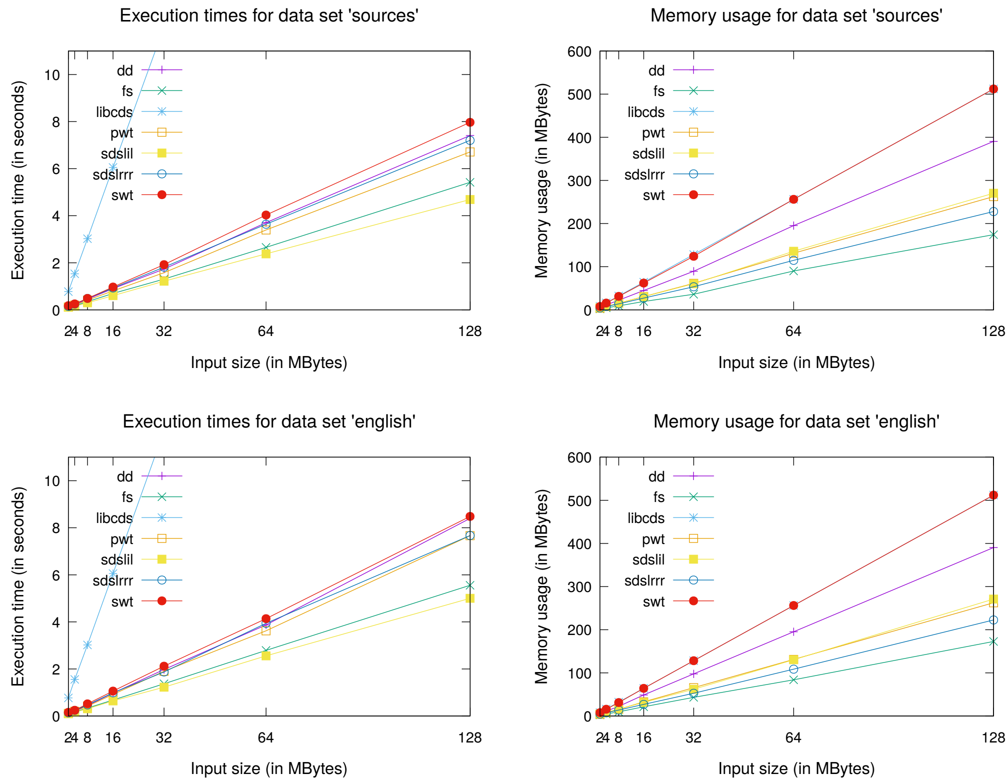


Figure 2 Experimental results (Part 2 – continued from previous page).

($\lg m = 4$) are roughly half of those for the english data set ($\lg m \approx 8$), with the notable exception of libcds. The results confirm the expected $O(n \lg m)$-time behaviour for (almost) all algorithms, although naturally not always with the same constant factors. Globally, all algorithms performed similarly (except libcds). sdslil was actually faster than the others, followed closely by our implementation.

In order to confirm the scalability of the method, we performed one last round of experiments with the largest input of Table 1, that is the 2.1GB english file. This time, we did not include the libcds method because it takes unreasonable time. As expected, the trend was maintained and sdslil was the fastest at 1m23s, followed by fs 1m31s. Other times were significantly higher: pwt 2m01s, sdslrrr 2m07s, dd 2m12s, and swt 2m39s.

5.2 Memory experiments

We also measured the memory usage of all algorithms on each input file using the `massif` tool of the `valgrind` package [22]. Since all algorithms are deterministic, the memory consumption does not vary between executions with the same input. The results are summarised in the right column of Figure 2. Shown are the peak total heap usage in MB during each execution.

As with time, there is a clear linear dependence on the text length for all algorithms, for any given alphabet size. The logarithmic dependence on the alphabet size is still clearly visible for some, in particular fs, but not all implementations. It is important to observe that the results shown in Figure 2 comprise the space taken by the WT itself, and the extra working space used by the construction algorithm. The fact that some implementations, like

Table 2 Average memory overheads (standard deviation in parenthesis).

Data set	Algorithm			
	dd	fs	libcds	pwt
dna	2.76 (0.25)	0.34 (0.04)	6.27 (0.02)	1.90 (0.13)
proteins	2.84 (0.00)	0.33 (0.04)	5.12 (0.02)	1.79 (0.00)
xml	2.39 (0.00)	0.37 (0.03)	3.41 (0.01)	1.33 (0.00)
sources	1.93 (0.12)	0.24 (0.11)	3.09 (0.01)	1.00 (0.06)
english	1.98 (0.13)	0.29 (0.07)	3.06 (0.01)	1.02 (0.07)
	sdslil	sdslrrr	swt	
dna	0.99 (0.32)	0.72 (0.28)	5.88 (0.11)	
proteins	1.39 (0.30)	1.28 (0.30)	5.12 (0.02)	
xml	1.33 (0.22)	1.15 (0.22)	3.71 (0.02)	
sources	1.10 (0.16)	0.85 (0.16)	3.00 (0.05)	
english	1.11 (0.14)	0.80 (0.15)	3.01 (0.05)	

libcds and swt, use much more space than the others, while representing the same information, and, moreover, that this working memory varies very little with the alphabet size, suggest that the total space is dominated by structures used by the construction algorithm which depend essentially on n.

We also estimated the relative *space overhead*, defined as the ratio $(M - S)/S$, where $S = m \lg n$ bits is a lower bound of the space taken by the uncompressed WT, and M is the total memory measured in the experiments. The average overheads per algorithm and per data set are shown in Table 2. The numbers confirm that the extra memory required by fs was comfortably under the predicted upper bound of 50% due to its use of a $\alpha = 1.5$ growth factor for the dynamic bit arrays. In all tests, our implementation outperformed the others in that respect, seconded only by sdslrrr, which uses compression, with 2–3 times more overhead.

Finally, we also performed one last test to gauge memory consumption using the 2.1GB english file. Our method performed better than all the others by similar margins. The fs implementation used 2.80GB, seconded by sdslrrr, which used 3.57GB, followed by pwt 4.22GB, sdslil 4.34GB, dd 6.27GB, and swt 8.23GB.

6 Discussion

We have presented a method for the online construction of the balanced wavelet tree of a source text T requiring $O(n \lg m)$ time and very little working memory. No previous information about the alphabet is assumed. We argue that our algorithm is conceptually simpler than most other methods but, despite its simplicity, it compares quite well in practice against other implementations, as shown by a series of experiments on real data, offering an appealing time vs. space compromise, not to mention that the online characteristic makes it more amenable to certain applications.

We regret not having an implementation of [13] to compare, but we note that, even for inputs of one petabyte (2^{50} bytes), the theoretical speedup would be of just about $7\times$. This margin can be easily eroded in practice by a more complex code. We equally regret not having found the space-efficient implementation `PERMUTE` [3] although the asymptotic costs and the reported comparisons of this algorithm against LIBCDS suggest that our implementation could still be a more favourable compromise.

Our current implementation is pointer-based, requiring $O(m \lg n)$ bits of space for the tree topology alone, which can be a drawback for applications with large alphabets. To mitigate this problem, we notice that Algorithm 2 builds the WT one level at a time, and so we can represent its structure implicitly, at any time, by using an array $P[0:h]$ of $h = 2^{\lceil \lg s \rceil} - 1$ positions, where s stands for the number of characters currently represented. As in the binary heap [5, Sec 6.1], we assign the root to position 0 and, from there on, the left and right children of the node at position i are associated to positions $2i + 1$ and $2i + 2$, respectively. We let $P[i]$ store the (pointer to the) bit vector of the node corresponding to position i. In this case, P, and hence the WT, is actually a dynamic array of bit vectors which is doubled every time a node is first added to a new level. This way we still have constant amortised time per node creation and we no longer need the node pointers. However we may yet have up to $s - 2$ unused positions of P, corresponding to the incomplete lowest level of the WT. So, the overall space is about the same in the worst case as with node pointers, but it may be more space-efficient in practice since we expect the lowest level to be fairly populated, and the unused positions at the end of P can be easily trimmed off after the construction.

Notice also that, in the test data sets of Table 1, the alphabet size was always under 256, so that it could be efficiently represented as a simple byte array, whereas in applications with large alphabets, the representation of the alphabet itself can be an issue. However, this is arguably a separate problem not particular to our construction method. In fact Algorithm 2 is reasonably oblivious to the actual data structure used, other than by supposing that it is a form of dictionary that supports insertion and membership queries in constant time. Notice that the space required by the alphabet was consciously excluded from the space complexity of Proposition 3.

Another limitation of our method is that it imposes an order on the alphabet symbols by somehow sorting them according to their order of appearance on the represented string. By contrast, the traditional balanced WT construction recursively partitions the alphabet in halves, respecting a certain 'natural' order, e.g. the lexicographic order for character alphabets, or the ascending order for integer alphabets. While our approach is coherent with the interpretation that the alphabet is 'unknown' (and hence so is its natural order), and this does not present a problem for the rank, select, or access operations, it may void the use of the WT on applications that assume a specific order for the alphabet, like in range quantile queries [8].

The first algorithm presented in this paper can be adapted to other WT topologies like the Huffman WT [18]. If we know the alphabet and relative character frequencies, then we can build the template tree following the greedy $O(m \lg m)$ Huffman tree construction algorithm [5, Sec 16.3] and then fill its contents just like in Algorithm 1. Even if the alphabet is unknown, we can still first build the template with one pass over T and then fill it in a second pass. It remains to be shown whether the single-pass online method could be adapted to the Huffman WT within the same time bounds.

Finally, we remark that the online construction procedure shown in Algorithm 2 suggests the static case where the input string/stream is read to its end to fill the raw bit contents of the nodes, before the WT is ever used. These raw bit vectors are then usually processed in linear time to produce a sub-linear amount of supporting information that allow for constant-time binary rank and select queries. However, we notice that the construction method is independent of the actual bit vector implementation. All it supposes is that the bit vectors support constant (amortised) time append operations so that Proposition 3 holds. In particular, if dynamic rank/select bit vectors are used [16, 4] then this construction method yields a partially dynamic WT implementation that allows for rank, select, and access operations to be performed at any time on a prefix of the input sequence.

7 Availability

The source code, as well as the experimental data and scripts used in this paper can be obtained from `http://www.cin.ufpe.br/~paguso/sea2017`.

Acknowledgments. We thank the anonymous reviewers for their thoughtful comments.

References

1. David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
2. Francisco Claude. LIBCDS: A compressed data structures library (https://github.com/fclaude/libcds2). URL: `https://github.com/fclaude/libcds2`.
3. Francisco Claude, Patrick K. Nicholson, and Diego Seco. Space Efficient Wavelet Tree Construction. In *Proceedings of the 18th Symposium on String Processing and Information Retrieval - SPIRE 2011*, pages 185–196, Pisa, 2011. `doi:10.1007/978-3-642-24583-1_19`.
4. Joshimar Cordova and Gonzalo Navarro. Practical Dynamic Entropy-Compressed Bitvectors with Applications. In *Proceedings of the 15th International Symposium on Experimental Algorithms – SEA 2016*, pages 105–117, 2016. `doi:10.1007/978-3-319-38851-9_8`.
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, 1990.
6. Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of Wavelet Trees. *Information and Computation*, 207(8):849–866, 2009. `doi:10.1016/j.ic.2008.12.010`.
7. José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Efficient Wavelet Tree Construction and Querying for Multicore Architectures. In *Proceedings of the 13th International Symposium on Experimental Algorithms – SEA 2014*, pages 150–161, Copenhagen, 2014. `doi:10.1007/978-3-319-07959-2_13`.
8. Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range Quantile Queries: Another Virtue of Wavelet Trees. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval – SPIRE 2009*, pages 1–6, Saariselkä, 2009. `arXiv:arXiv:0903.4726v6`, `doi:10.1007/978-3-642-03784-9_1`.
9. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proceedings of the 13th International Symposium on Experimental Algorithms – SEA 2014*, pages 326–337, Copenhagen, 2014. `doi:10.1007/978-3-319-07959-2_28`.
10. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of the 14th annual ACM-SIAM Symposium On Discrete Algorithms – SODA 2003*, pages 841–850, Philadelphia, 2003.
11. J. Ian Munro. Tables. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, Hyderabad, 1996. `doi:10.1007/3-540-62034-6_35`.
12. J. Ian Munro. Personal communication, 2017.
13. J. Ian Munro, Yakov Nekrich, and Jeffrey S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. `doi:10.1016/j.tcs.2015.11.011`.
14. ISO/IEC. Information technology – Programming languages – C. *ISO/IEC 9899:2011 Std*, 2011.
15. Guy Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1989.

16 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):1–38, 2008. doi:10.1145/1367064.1367072.

17 Christos Makris. Wavelet trees: A survey. *Computer Science and Information Systems*, 9(2):585–625, 2012. doi:10.2298/CSIS110606004M.

18 Gonzalo Navarro. Wavelet Trees for All. In *Proceedings of the 23rd Annual conference on Combinatorial Pattern Matching – CPM 2012*, pages 2–26, Helsinki, 2012. doi:10.1007/978-3-642-31265-6_2.

19 Gonzalo Navarro. *Compact data structures: a pratical approach*. Cambridge Univ Press, 2016.

20 Gonzalo Navarro and Paolo Ferragina. Pizza&Chili Corpus Website (http://pizzachili.dcc.uchile.cl/). URL: http://pizzachili.dcc.uchile.cl/.

21 Gonzalo Navarro and Eliana Providel. Fast, Small, Simple Rank/Select on Bitmaps. In *Proceedings of the 11th international Symposium on Experimental Algorithms – SEA 2012*, pages 295–306, Bordeaux, 2012. doi:10.1007/978-3-642-30850-5_26.

22 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation – PLDI 2007*, pages 89–100, New York, 2007. doi:10.1145/1250734.1250746.

23 Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms – SODA 2002*, pages 233–242, San Francisco, 2002. arXiv:arXiv:0705.0552v1.

24 Julian Shun. Parallel Wavelet Tree Construction. In *2015 Data Compression Conference*, pages 63–72. IEEE, apr 2015. doi:10.1109/DCC.2015.7.

25 German Tischler. On wavelet tree construction. In *Proceedings of the 22nd annual conference on Combinatorial pattern matching – CPM 2011*, pages 208–218, Palermo, 2011.

Engineering External Memory LCP Array Construction: Parallel, In-Place and Large Alphabet

Juha Kärkkäinen[1] and Dominik Kempa[2]

1 **Department of Computer Science, University of Helsinki, Helsinki, Finland**
`juha.karkkainen@cs.helsinki.fi`

2 **Helsinki Institute for Information Technology HIIT, Helsinki, Finland; and Department of Computer Science, University of Helsinki, Helsinki, Finland**
`dominik.kempa@cs.helsinki.fi`

Abstract

The suffix array augmented with the LCP array is perhaps the most important data structure in modern string processing. There has been a lot of recent research activity on constructing these arrays in external memory. In this paper, we engineer the two fastest LCP array construction algorithms (ESA 2016) and improve them in three ways. First, we speed up the algorithms by up to a factor of two through parallelism. Just 8 threads is sufficient for making the algorithms essentially I/O bound. Second, we reduce the disk space usage of the algorithms making them in-place: The input (text and suffix array) is treated as read-only and the working disk space never exceeds the size of the final output (the LCP array). Third, we add support for large alphabets. All previous implementations assume the byte alphabet.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases LCP array, suffix array, external memory algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.17

1 Introduction

The suffix array [12, 3], a lexicographically sorted list of the suffixes of a text, is one of the most important data structures in modern string processing. It is frequently augmented with the longest-common-prefix (LCP) array, which stores the lengths of the longest common prefixes between lexicographically adjacent suffixes. Together they are the basis of powerful text indexes such as enhanced suffix arrays [1] and many compressed full-text indexes [13]. Modern textbooks spend dozens of pages in describing their applications, see e.g. [14, 11].

The construction of the two arrays is a bottleneck in many applications. There has been a lot of recent research on external memory construction of these data structures. Here we are interested in the construction of the LCP array given the suffix array and the text. The two fastest external memory algorithms for this task are currently EM-SΦ and EM-SI, recently introduced in [4]. In this paper, we improve EM-SΦ and EM-SI in several ways.

First, we modify both algorithms to use multiple threads during their execution. Parallelization does not reduce or speed up I/O as such, but it can speed up those stages that are dominated by computation rather than I/O. Cache misses in particular can be expensive enough to dominate I/O. Our experimental results show that EM-SI benefits only a little from parallelization, but the speed of EM-SΦ improves by as much as a factor of two for

© Juha Kärkkäinen and Dominik Kempa;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 17; pp. 17:1–17:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

some texts. Eight threads is sufficient to achieve an essentially maximum speed up as the computation becomes I/O bound.

Second, we reduce the disk space usage of both algorithms. Disk space usage can be more crucial than speed, because a lack of sufficient free disk space can prevent a computation entirely. We make both algorithms in-place in the sense that the disk space usage never exceeds what is needed for the input (the text, the suffix array, and for EM-SI, the Burrows–Wheeler transform (BWT)) and the output (the LCP array). Thus any machine that has sufficient disk space for the inputs and the outputs can run these algorithms. The input is treated as read-only, i.e., it is never deleted or written over even temporarily. The working disk space, i.e., disk space used in addition to the inputs, is reduced by more than a factor of two in some cases. The fully in-place computation slows down the algorithms but never more than 36% and often much less.

Third, we modify both algorithms to handle large alphabets. All previous implementations work only for the byte alphabet. While it is possible to split large characters into multiple bytes, construct suffix and LCP arrays for the resulting text over the byte alphabet, and then post-process to construct the desired arrays, this requires much more time and disk space than using algorithms that can handle large alphabets natively. We demonstrate this for EM-SΦ and EM-SI in our experiments.

Related work. Suffix array construction in external memory has a long history. The most recent addition is fSAIS [8], which is also the first implementation able to handle large alphabets natively. LCP array construction in external memory has been studied much less. It was first achieved by modifying suffix array construction to produce the LCP array simultaneously [2]. Independent construction of the LCP array given the suffix array as input is preferable and was first achieved by LCPScan [5]. The only further practical improvements are EM-SΦ and EM-SI. A very recent theoretical break-through is the first LCP array construction algorithm [6] with I/O complexity $\mathcal{O}(\mathrm{sort}(n))$ for a text of length n, where $\mathrm{sort}(n)$ is the complexity of sorting n integers, but it is not competitive in practice. Another recent result is an algorithm for computing the succinct representation of the PLCP (permuted LCP) array in external memory [16].

2 Basic Data Structures

Throughout we consider a string $\mathsf{X} = \mathsf{X}[0..n) = \mathsf{X}[0]\mathsf{X}[1]\ldots\mathsf{X}[n-1]$ of $|\mathsf{X}| = n$ symbols drawn from an alphabet of size σ. Here and elsewhere we use $[i..j)$ as a shorthand for $[i..j-1]$. For $i \in [0..n]$, we write $\mathsf{X}[i..n)$ to denote the *suffix* of X of length $n-i$, that is $\mathsf{X}[i..n) = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[n-1]$. We will often refer to suffix $\mathsf{X}[i..n)$ simply as "suffix i".

The *suffix array* [12, 3] of X is an array $\mathrm{SA} = \mathrm{SA}[0..n]$ which contains a permutation of the integers $[0..n]$ such that $\mathsf{X}[\mathrm{SA}[0]..n) < \mathsf{X}[\mathrm{SA}[1]..n) < \cdots < \mathsf{X}[\mathrm{SA}[n]..n)$. In other words, $\mathrm{SA}[j] = i$ iff $\mathsf{X}[i..n)$ is the $(j+1)^{\mathrm{th}}$ suffix of X in ascending lexicographical order. Another representation of the permutation is the *Φ array* [9] $\Phi[0..n)$ defined by $\Phi[\mathrm{SA}[j]] = \mathrm{SA}[j-1]$ for $j \in [1..n]$. In other words, the suffix $\Phi[i]$ is the immediate lexicographical predecessor of the suffix i, and thus $\mathrm{SA}[n-k] = \Phi^k[\mathrm{SA}[n]]$ for $k \in [0..n]$. An example illustrating the arrays is given in Table 1.

Let $\mathrm{lcp}(i,j)$ denote the length of the longest-common-prefix (LCP) of suffix i and suffix j. For instance, in the example of Table 1, $\mathrm{lcp}(0,6) = 3 = |\mathtt{bab}|$ and $\mathrm{lcp}(7,4) = 5 = |\mathtt{abbab}|$. The *longest-common-prefix array* [12, 10], $\mathrm{LCP}[1..n]$, is defined such that $\mathrm{LCP}[i] = \mathrm{lcp}(\mathrm{SA}[i], \mathrm{SA}[i-1])$ for $i \in [1..n]$. The *permuted LCP array* [9] $\mathrm{PLCP}[0..n)$ is the LCP array

Table 1 Examples of the arrays used by the algorithms for the text $\mathsf{X} = \texttt{babaabbabbab}$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$\mathsf{X}[i]$	b	a	b	a	a	b	b	a	b	b	a	b	-
$\mathrm{SA}[i]$	12	3	10	1	7	4	11	2	9	0	6	8	5
$\mathrm{BWT}[i]$	b	b	b	b	b	a	a	a	b	$	b	a	a
$\Phi[i]$	9	10	11	12	7	8	0	1	6	2	3	4	-
$\mathrm{LCP}[i]$	-	0	1	2	2	5	0	1	2	3	3	1	4
$\mathrm{PLCP}[i]$	3	2	1	0	5	4	3	2	1	2	1	0	-
$i + \mathrm{PLCP}[i]$	3	3	3	3	9	9	9	9	9	11	11	11	-
$2i + \mathrm{PLCP}[i]$	3	4	5	6	13	14	15	16	17	20	21	22	-
$\mathrm{PLCP}_4^{\mathrm{lo}}(i)$	3	2	1	0	5	4	3	2	1	0	0	0	-
$\mathrm{PLCP}_4^{\mathrm{lo}}(i)$	3	8	7	6	5	4	3	2	1	2	1	0	-

permuted from the lexicographical order into the text order, i.e., $\mathrm{PLCP}[\mathrm{SA}[j]] = \mathrm{LCP}[j]$ for $j \in [1..n]$. Then $\mathrm{PLCP}[i] = \mathrm{lcp}(i, \Phi[i])$ for all $i \in [0..n)$. Table 1 shows example LCP and PLCP arrays.

The row $i + \mathrm{PLCP}[i]$ in Table 1 illustrates (the first part of) the following property of the PLCP array, which is the basis of all efficient algorithms for LCP array construction.

▶ **Lemma 1** ([5]). *Let $i, j \in [0..n)$. If $i \le j$, then $i + \mathrm{PLCP}[i] \le j + \mathrm{PLCP}[j]$. Symmetrically, if $\Phi[i] \le \Phi[j]$, then $\Phi[i] + \mathrm{PLCP}[i] \le \Phi[j] + \mathrm{PLCP}[j]$.*

The *succinct PLCP array* [15] $\mathrm{PLCP}_{\mathrm{succ}}[0..2n)$ represents the PLCP array using $2n$ bits. Specifically, $\mathrm{PLCP}_{\mathrm{succ}}[j] = 1$ if $j = 2i + \mathrm{PLCP}[i]$ for some $i \in [0..n)$, and $\mathrm{PLCP}_{\mathrm{succ}}[j] = 0$ otherwise. Notice that the value $2i + \mathrm{PLCP}[i]$ is unique for each i by Lemma 1 as illustrated in Table 1.

For $q \ge 1$, the *sparse PLCP array* $\mathrm{PLCP}_q[0..\lceil n/q \rceil)$ is defined by $\mathrm{PLCP}_q[i] = \mathrm{PLCP}[iq]$, i.e., it contains every q^{th} entry of PLCP. We also define $\Phi_q[0..\lceil n/q \rceil)$ by $\Phi_q[i] = \Phi[iq]$ so that $\mathrm{PLCP}_q[i] = \mathrm{lcp}(qi, \Phi_q[i])$. The sparse PLCP array can be used as a compact representation of the full PLCP array because the other entries can be bounded using the following lemma.

▶ **Lemma 2** ([9]). *For any $i \in [0..n)$, let*

$$\mathrm{PLCP}_q^{\mathrm{lo}}(i) = \max(0, \mathrm{PLCP}_q[\lfloor i/q \rfloor] - (i - q\lfloor i/q \rfloor))$$

$$\mathrm{PLCP}_q^{\mathrm{hi}}(i) = \begin{cases} \mathrm{PLCP}_q[\lceil i/q \rceil] + (q\lceil i/q \rceil - i) & \text{if } q\lceil i/q \rceil < n \\ n - i - 1 & \text{otherwise} \end{cases}$$

Then $\mathrm{PLCP}_q^{\mathrm{lo}}(i) \le \mathrm{PLCP}[i] \le \mathrm{PLCP}_q^{\mathrm{hi}}(i)$.

Although the difference $\mathrm{PLCP}_q^{\mathrm{hi}}(i) - \mathrm{PLCP}_q^{\mathrm{lo}}(i)$ has no non-trivial limit for an individual i, the sum of the differences is bounded by the following lemma.

▶ **Lemma 3** ([9]). $\sum_{i \in [0..n)} \mathrm{PLCP}_q^{\mathrm{hi}}(i) - \mathrm{PLCP}_q^{\mathrm{lo}}(i) \le (q-1)n + q^2$.

The *Burrows–Wheeler transform* $\mathrm{BWT}[0..n]$ of X is defined by $\mathrm{BWT}[i] = \mathsf{X}[\mathrm{SA}[i] - 1]$ if $\mathrm{SA}[i] > 0$ and otherwise $\mathrm{BWT}[i] = \$$, where $\$$ is a special symbol that does not appear in the text. We say that an lcp value $\mathrm{LCP}[i] = \mathrm{PLCP}[\mathrm{SA}[i]]$ is *reducible* if $\mathrm{BWT}[i] = \mathrm{BWT}[i-1]$ and *irreducible* otherwise. The significance of reducibility is summarized in the following two lemmas.

▶ **Lemma 4** ([9]). *If $\mathrm{PLCP}[i]$ is reducible, then $\mathrm{PLCP}[i] = \mathrm{PLCP}[i-1] - 1$ and $\Phi[i] = \Phi[i-1] + 1$.*

▶ **Lemma 5** ([9, 7]). *The sum of all irreducible lcp values is $\le n \log n$.*

3 Basic Algorithms

In this section, we describe the basic algorithms EM-SΦ and EM-SI introduced in [4]. Some details are omitted and others only sketched; we refer to [4] for full details.

3.1 EM-SΦ Algorithm

The first algorithm EM-SΦ gets the text X and the suffix array SA as input, and performs the following steps:

1. Compute PLCP_q for q chosen so that PLCP_q fits in RAM. During this step, we consider the text divided into *segments* that fit in RAM.
   a. Compute Φ_q by scanning SA.
   b. Generate all pairs $(i, \Phi[i])$ such that i is a multiple of q using Φ_q. Write each pair $(i, \Phi[i])$ to disk into the file associated with the text segment that contains $\Phi[i]$. Notice that the pairs in each file are naturally sorted by i.
   c. For each text segment, load the segment into RAM. Read the pairs $(i, \Phi[i])$ from the associated file while simultaneously scanning the full text so that the position $\mathsf{X}[i]$ is reached when the pair $(i, \Phi[i])$ is processed. For each pair, compute $\mathrm{lcp}(i, \Phi[i])$ and write it to disk into a separate file for each segment. When computing $\ell = \mathrm{lcp}(i, \Phi[i])$ we use the fact that $\ell \geq \mathrm{lcp}(i', \Phi[i']) - (i - i')$ by Lemma 1, where $(i', \Phi[i'])$ is the pair processed just previously. This ensures that the text scan never needs to backtrack.
   d. Construct PLCP_q in RAM by reading each value $\mathrm{PLCP}_q[i]$ from the file associated with the text segment containing $\Phi_q[i]$.
2. Compute LCP using PLCP_q based on Lemma 2. During this step, we consider the text divided into *half-segments* so that two half-segments fit in RAM. Every possible pair of half-segments is loaded into RAM once.
   a. Scan SA to generate all $(i, \Phi[i])$ pairs. For each pair use PLCP_q (stored in RAM) to compute the lower bound $\ell = \mathrm{PLCP}_q^{\mathrm{lo}}(i)$ (and the upper bound $\mathrm{PLCP}_q^{\mathrm{hi}}(i)$) for $\mathrm{PLCP}[i]$ using Lemma 2. Write the pair $(i + \ell, \Phi[i] + \ell)$ to disk, where there is a separate file for each pair of half-segments. If $\mathrm{PLCP}_q^{\mathrm{lo}}(i) = \mathrm{PLCP}_q^{\mathrm{hi}}(i)$, no pair is written since we already know the exact lcp value.
   b. For each pair of half-segments, load them to RAM and compute $\mathrm{lcp}(j, k)$ for each pair (j, k) obtained from the associated file. The resulting value $\mathrm{lcp}(j, k)$ is written to disk to a separate file for each pair of half-segments.
   c. Scan SA to generate all $(i, \Phi[i])$ pairs. For each pair, compute $\ell = \mathrm{PLCP}_q^{\mathrm{lo}}(i)$ (and $\mathrm{PLCP}_q^{\mathrm{hi}}(i)$) as in step 2(a) and read the value $\ell' = \mathrm{lcp}(i + \ell, \Phi[i] + \ell)$ from the appropriate file. Then $\mathrm{PLCP}[i] = \ell + \ell'$ is the next value in the LCP array.

In Step 1(c), the computation of the lcp value may overflow the segment, i.e., $\Phi[i]$ is in the segment but $\Phi[i] + \ell$ is not. To deal with an overflow, we have in RAM an overflow buffer (of size of one disk block) containing the beginning of the next segment. An overflow beyond even the overflow buffer is handled by reading from disk. Similar overflows can occur in Step 2(b). There too we use overflow buffers but comparisons beyond the overflow buffers are simply aborted. In cases, where such an aborted comparison is possible (based on the upper bound $\mathrm{PLCP}_q^{\mathrm{hi}}(i)$), extra pairs are generated in Step 2(a) to continue the potentially aborted comparison when the appropriate pair of half-segments is in RAM. We refer to [4] for further details and analysis of overflow handling.

3.2 EM-SI Algorithm

The second algorithm EM-SI gets X, SA and BWT as input, and has the following steps:

1. Compute the succinct PLCP array $\mathrm{PLCP}_{\mathrm{succ}}$. For this step, we consider $\mathrm{PLCP}_{\mathrm{succ}}$ divided into segments that fit in RAM and the text divided into half-segments so that two half-segments fit in RAM.
   a. Scan SA and BWT to form a pair $(i, \Phi[i])$ for each i such that $\mathrm{PLCP}[i]$ is irreducible. The pairs are written to disk where there is a separate file for each pair of text half-segments.
   b. Compute the bitvector $\mathsf{R}[0..n]$, where $\mathsf{R}[i] = 1$ iff $\mathrm{PLCP}[i]$ is irreducible. If R does not fit in RAM, it is computed one RAM-sized segment at a time. The irreducible positions are determined either by scanning SA and BWT or by scanning the irreducible $(i, \Phi[i])$ pairs produced in Step 1(a), whichever takes less I/O.
   c. For each pair of text half-segments, load them to RAM and compute $\mathrm{PLCP}[i] = \mathrm{lcp}(i, \Phi[i])$ for each pair $(i, \Phi[i])$ obtained from the associated file. For each computed $\mathrm{PLCP}[i]$, we write the value $2i + \mathrm{PLCP}[i]$ to disk into a separate file for each $\mathrm{PLCP}_{\mathrm{succ}}$ segment.
   d. For each $\mathrm{PLCP}_{\mathrm{succ}}$ segment, initialize it with zeros in RAM, read the values from the associated file and set the corresponding bits to 1. Then read the corresponding part of R to determine the reducible lcp values using Lemma 4 and set the corresponding bits of $\mathrm{PLCP}_{\mathrm{succ}}$ to 1. See [4] for details.
2. Compute LCP from $\mathrm{PLCP}_{\mathrm{succ}}$. For this step, we consider the full PLCP array (not $\mathrm{PLCP}_{\mathrm{succ}}$) divided into segments that fit into RAM.
   a. Scan SA and write each value $\mathrm{SA}[i]$ to disk into the file associated with the PLCP segment that contains the position $\mathrm{SA}[i]$.
   b. For each PLCP segment, create it in RAM by scanning the relevant part of $\mathrm{PLCP}_{\mathrm{succ}}$. Then read the $\mathrm{SA}[i]$ values from the associated file, compute $\mathrm{LCP}[i] = \mathrm{PLCP}[\mathrm{SA}[i]]$, and write it to disk into a separate file for each segment.
   c. Scan SA, and for each $i \in [0..n)$, read $\mathrm{LCP}[i]$ from the file associated with the PLCP segment that contains $\mathrm{SA}[i]$ and write it to the final LCP file.

Overflows in Step 1(c) are handled as in Step 1(c) of EM-SΦ: using overflow buffers, and when that is not enough, reading directly from disk.

4 Parallelization

We have implemented both algorithms to use multiple threads during most stages of the computation. In both algorithms, several stages process a sequence of items so that the computation for one item is independent of other items and takes approximately the same time for all items. Such computation is trivial to parallelize: load a bufferful of items at a time to RAM and split the buffer evenly between threads. Below we describe the parallelization only for more complicated stages.

4.1 Parallelizing EM-SΦ

The first more complicated stage in EM-SΦ is Step 1(c). Here the algorithm processes a sequence of $(i, \Phi[i])$ pairs for each text segment, and the computation for each pair depends on the preceding pair. In the parallel version, the full sequence of pairs on disk is evenly split among the threads, and each thread processes its part completely independently from other threads.

During the stage, each thread has to scan a part of the text. Typically, each thread scans a different part of the text with only a negligible overlap between the parts. However, for highly repetitive texts with extremely large lcp values the overlaps can be large which could increase the amount of I/O significantly. To avoid this, we compute a super-sparse PLCP array PLCP_p for $p \gg q$, which can be done quickly using essentially the internal memory Φ-algorithm [9]. We then use PLCP_p to compute lower bounds according to Lemma 2, which limits the total overlaps to less than n. The sizes of the text parts for different threads may vary but this is no problem as text scanning is strongly I/O-bound. The important thing is to minimize the times when no thread is doing I/O.

The second nontrivial parallelization in EM-SΦ is Step 2(b). Here processing a single item, i.e., computing $\mathrm{lcp}(j,k)$, can involve a very long string comparison in RAM. The length of each comparison is not known in advance which makes load balancing difficult. Very long string comparisons are rare even for highly repetitive texts, but they tend to come in clusters. To see why this happens, consider a pair $(j,k) = (i+\ell, \Phi[i]+\ell)$, where $\ell = \mathrm{PLCP}_q^{\mathrm{lo}}(i)$. The lower bound ℓ ensures that the *average* length of comparisons is less than q, but there can still be rare cases where the lower bound is poor and a long comparison results. If (j,k) is such a case, then so is $(j',k') = ((i+1)+\mathrm{PLCP}_q^{\mathrm{lo}}(i+1), \Phi[i+1]+\mathrm{PLCP}_q^{\mathrm{lo}}(i+1))$ unless $i+1$ is is a multiple of q. If furthermore $i+1$ is a reducible position – and most positions are reducible for highly repetitive texts – $k' = k$ and $j' = j$ or $k' = k+1$ and $j' = j+1$, and thus (j,k) and (j',k') are processed with the same pair of half-segments. If $i+2$, $i+3$ and so on are reducible positions too, we can get a cluster of such bad cases. We could identify such a cluster by the fact that the difference $k-j$ is the same for all the pairs, and once identified, we can use Lemma 4 to avoid doing a long comparison more than once. However, the identification of such a cluster would require sorting the pairs (j,k) by $k-j$, and avoiding expensive sorting was one of the main ideas of the original algorithm. Consequently, our approach is to first ignore potential long comparisons and simply split a bufferful of (j,k) pairs evenly between threads. However, the average length of string comparisons is monitored, and if it exceeds a threshold, the computation is aborted and the buffer is processed in a long-lcp mode instead. In the long-lcp mode, the (j,k) pairs in the buffer are sorted by the difference $k-j$ so that we can then utilize Lemma 4. The sorting is parallelized, and the sorted buffer is split evenly among threads. Because of the sorting, the long-lcp mode is too slow to use all the time. This approach speeds up the computation significantly for highly repetitive files even in a single thread mode, and with mutiple threads it avoids bad load balancing.

4.2 Parallelizing EM-SI

The first nontrivial step in EM-SI is Step 1(c). As in Step 2(b) of EM-SΦ, some string comparisons can be long, but similar clustering of long comparisons is very unlikely because only irreducible lcp values are computed. Thus simply splitting a bufferful of $(i, \Phi[i])$ pairs evenly between threads works well enough and there is no monitoring of comparison lengths. However, the exception are comparisons that extend beyond the end of a half-segment and even the overflow buffers (which never happens in Step 2(b) in EM-SΦ). In the sequential version, such a comparison is completed immediately by reading parts of text from disk. In the parallel version, the extended comparisons are postponed until all threads are finished, and then performed separately.

The second nontrivial step in EM-SI is Step 1(d), where we want to set bits in a bitvector held in RAM (a segment of $\mathrm{PLCP}_{\mathrm{succ}}$) at positions read from disk. The problem in parallelizing this arises from two or more threads trying to simultaneously set different bits

in the same byte or word. To avoid this, the bitvector is divided into buckets and a bufferful of positions to set is first distributed into the buckets. Then each bucket is processed by a single thread. There are more buckets than threads and the buckets are assigned to threads so that each thread sets about the same number of positions.

5 In-Place Computation

When determining the disk usage of the algorithms, we assume that the inputs are on disk during the whole computation and are never modified. We are then interested in the disk space used in addition to the input, which we call the *working disk space*. The smallest possible working disk space by any algorithm is the size of the output, the LCP array. In this section, we describe modifications to the algorithms that achieve exactly this minimum working space making the algorithms in-place in this sense.

For concreteness, we assume that large integers, including lcp values, are stored using 40-bit integers by default, as they are in our current implementation, and thus the minimum working space is $40n$ bits or $5n$ bytes.

The peak working disk space usage in basic EM-SΦ happens in Step 2(b) and is $15n$ bytes in the worst case consisting of n (j, k) pairs and n lcp values produced as output of the step. In practice, the peak is closer to $10n$ bytes since each file of (j, k) pairs is deleted when it is no more needed, and can be even less because no (j, k) pair is stored when $\mathrm{lcp}(i, \Phi[i])$ can be determined from the sparse PLCP array. A working disk space of $10n$ bytes may be needed in Step 2(c) too.

In EM-SI, the worst case peak disk usage can be $15n$ bytes in Step 1(c). However, the disk usage is actually 10–$15r$ bytes, where r is the number of irreducible lcp values. For many files, $r < n/3$ and the disk usage of Step 1(c) is actually less than $5n$ bytes. In Step 2(c), the disk usage is always $10n$ bytes without any optimization.

Both algorithms involve large files that are scanned once and then deleted. In our implementation of the basic algorithms, such files are split into multiple subfiles that are deleted as soon as the scan has passed them. This reduces the working disk space of both algorithms to about $10n$ bytes in the worst case and just slightly more than $5n$ bytes is some cases. There are realistic inputs requiring about $10n$ bytes, which is still twice the minimum, and our goal is to reduce it to $5n$ bytes in all cases.

5.1 Compact Encoding of LCP Values

By default, lcp values are stored on disk using 5 bytes, but in some cases we can reduce the space using special representations. One such special representation is used for storing the sparse PLCP array PLCP_q. The default representation needs $5n/q$ bytes, but when $q < 40$ we instead use a bitvector of $n + n/q$ bits defined similarly to $\mathrm{PLCP}_{\mathrm{succ}}$.

The main technique to reduce the size of lcp values is the V-byte encoding [17], which uses a variable number of bytes to store each value. The total size of such encoding can be bounded by the following result.

▶ **Lemma 6.** *The total number of bytes in the V-byte encoding of a sequence of $\leq k$ non-negative integers summing up to $\leq s$ is at most*

$$\begin{cases} k + s/2^7 & \text{if } s/k \leq 2^7 \\ 2k + (s - 2^7 k)/2^{15} & \text{if } 2^7 \leq s/k \leq 2^{15} + 2^7 \end{cases}$$

Table 2 Working disk space (in bytes) during Steps 2(a) and (b) in EM-SΦ with partitioning.

i	n_i/n	q	pairs	lcp values	PLCP_q	total
1	0.395	4..39	$3.95n$	$n_1 + qn/2^7$	$\frac{n+n/q}{8}$	$\left(4.47 + \frac{q}{2^7} + \frac{1}{8q}\right) n < 4.78n$
		40..50	$3.95n$	$n_1 + qn/2^7$	$5n/q$	$\left(4.345 + \frac{q}{2^7} + \frac{5}{q}\right) n < 4.83n$
		51..8551	$3.95n$	$2n_1 + \frac{qn-2^7 n_1}{2^{15}}$	$5n/q$	$\left(4.74 + \frac{q}{2^{15}} - \frac{0.395}{2^8} + \frac{5}{q}\right) n < 5n$
2	0.330	4..39	$3.3n$	$n_1 + n_2 + qn/2^7$	$\frac{n+n/q}{8}$	$\left(4.15 + \frac{q}{2^7} + \frac{1}{8q}\right) n < 4.46n$
		40..92	$3.3n$	$n_1 + n_2 + qn/2^7$	$5n/q$	$\left(4.025 + \frac{q}{2^7} + \frac{5}{q}\right) n < 4.8n$
		93..8264	$3.3n$	$2(n_1+n_2) + \frac{qn-2^7(n_1+n_2)}{2^{15}}$	$5n/q$	$\left(4.75 + \frac{q}{2^{15}} - \frac{0.725}{2^8} + \frac{5}{q}\right) n < 5n$
3	0.275	4..39	$2.75n$	$n + qn/2^7$	$\frac{n+n/q}{8}$	$\left(3.875 + \frac{q}{2^7} + \frac{1}{8q}\right) n < 4.19n$
		40..127	$2.75n$	$n + qn/2^7$	$5n/q$	$\left(3.75 + \frac{q}{2^7} + \frac{5}{q}\right) n < 4.79n$
		128..8300	$2.75n$	$2n + \frac{qn-2^7 n}{2^{15}}$	$5n/q$	$\left(4.75 + \frac{q}{2^{15}} - \frac{1}{2^8} + \frac{5}{q}\right) n < 5n$

The output of Step 2(b) of EM-SΦ consists of the values $\mathrm{PLCP}[i] - \mathrm{PLCP}_q^{\mathrm{lo}}(i)$, which we call lcp delta values. By Lemma 3, the sum of all n lcp delta values is at most qn, and thus we can use Lemma 6 to bound the total size. The details are described in Section 5.2.

To take advantage of V-byte encoding in EM-SI, we make some modifications to it. First, in Step 1(c), we write $\mathrm{PLCP}[i]$ instead of $2i + \mathrm{PLCP}[i]$ to output and use V-byte encoding. Since the total sum of irreducible lcp values is at most $n \log n$, we can again bound the total size by Lemma 6. Instead of deleting the pairs $(i, \Phi[i])$ as soon as possible, we keep the i's so that we can compute $2i + \mathrm{PLCP}[i]$ in Step 1(d). To be able to delete the $\Phi[i]$'s earlier, they are stored in a different file than the i's.[1]

The second modification to EM-SI is in Step 2, where we now construct and use a sparse PLCP array PLCP_q that fits in RAM. In the output of Step 2(b) and input of Step 2(c), we replace each value $\mathrm{LCP}[i]$ with the corresponding lcp delta value $\mathrm{LCP}[i] - \mathrm{PLCP}_q^{\mathrm{lo}}(\mathrm{SA}[i])$. Then we can again use V-byte encoding and Lemma 6 to bound the total size of the lcp values.

5.2 Partitioning

The main tool for reducing disk space usage is a technique called *partitioning* introduced in [5]. Consider Step 2(a) in EM-SΦ that produces and stores up to n (j,k) pairs and then Step 2(b) processes and deletes the pairs. The pairs need up to $10n$ bytes of temporary disk space. To reduce the space, we divide the pairs into three parts of sizes $n_1 = 0.395n$, $n_2 = 0.33n$ and $n_3 = 0.275n$, and perform the Steps 2(a) and (b) for one part at a time. Then the peak disk usage stays under $5n$ bytes at all times as detailed in Table 2. There is an upper limit of 8264 on the value of q but that is sufficient to fit PLCP_q into RAM in all practical scenarios. Furthermore, a larger q or smaller disk usage can be achieved by using more than three parts.

We also use partitioning in Step 1 of EM-SI. We perform the full step 1 except the setting of reducible bits for one part at a time. That is, after processing one part, we will have, for each irreducible i processed in that part, the bit $2i + \mathrm{PLCP}[i]$ set in $\mathrm{PLCP}_{\mathrm{succ}}$ and the bit i set in R. Once all parts have been processed, we produce the final $\mathrm{PLCP}_{\mathrm{succ}}$ by setting the reducible bits. With three parts of size at most $n/3$ each, the working disk space stays well under $5n$ bytes.

[1] The separation of i's and $\Phi[i]$'s into separate files helps with Step 1(b) too, because we need only i's to determine the irreducible positions.

The disadvantage of partitioning is that it needs some additional I/O depending on exactly how the partitioning is done. We have implemented two partitioning modes for each algorithm and always choose the one that produces less additional I/O. The first mode is called lex-partitioning and simply involves splitting SA into parts. When processing a part, we only need to scan the relevant part of SA. However, then each pair of text half-segments needs to be loaded into RAM once for each part. For large files, the loading of the half-segments dominates the I/O, and thus we instead use the second partitioning mode called text-partitioning. The items are partitioned according to which pair of half-segments they belong to. Then most pairs of half-segments need to be loaded only once. On the other hand, we then need to scan the full suffix array for each part, which makes it the slower option for smaller files.

Finally, partitioning also happens in Steps 2(a)–(b) in EM-SI. Recall that earlier we modified Step 2(b) to produce lcp delta values as output. In this case, we always perform lex-partitioning into two parts of sizes $n_1 = 0.58$ and $n_2 = 0.42$. Then the working disk space remains below $5n$ bytes when $q < 20000$.

5.3 Final Steps

The final step of EM-SΦ, Step 2(c), performs two tasks: it reads the lcp values from multiple files and merges them into a single sequence, and it converts the lcp delta values into the final lcp values. The in-place version separates the tasks, first merging in Step 2(c'), and then converting in Step 2(c"). Since the total size of the V-byte encoded delta values is always less than $2.5n$ bytes, the working disk space during merging stays below $5n$ bytes. For conversion, the merged delta value file is split into multiple subfiles so that each subfile can be deleted after it has been processed. The split points are decided adaptively during merging so that the working disk space never exceeds $5n$ bytes. The first subfile can always contain more than half the values. The following subfiles are smaller, and the last subfile is small enough so that we can load it in RAM and delete it from disk before conversion.

The final step of EM-SI, Step 2(c), after the modification to use V-byte encoded delta values, is essentially the same as the last step of EM-SΦ: merge and convert lcp delta values into the final LCP array, and is implemented similarly.

6 Experimental Results

Algorithms. The starting point and the baseline in our experiments are the original C++ implementations of the EM-SΦ and EM-SI algorithms described in [4]. Both algorithms are sequential, assume byte alphabet, and use more disk space than is needed for the output. We modified these implementations in the following ways:

- First, we parallelized the computation as described in Section 4. All basic parallelizations were done using OpenMP, and for more non-trivial parallelizations we use threads and synchronization mechanisms from the standard C++ library;
- Second, we added the "in-place mode" to both algorithms that reduces the working disk space to the space needed by the final LCP array as described in Section 5. We kept the "out-of-place mode" in the implementation for cases, where speed is the priority;
- Third, we modified the implementations to handle symbols of arbitrary size (that is a multiple of byte). This is relatively straightforward, as both algorithms only perform symbol comparisons, but our implementations are the first to explicitly support large-alphabet external-memory construction of the LCP array.

Table 3 Statistics of data used in the experiments; $100r/n$ is the percentage of irreducible lcp values among all lcp values (where r denotes the number of irreducible lcps) and Σ_r/r is the average length of the irreducible lcp value (where Σ_r is the sum of all irreducible lcps). Smaller files in experiments are prefixes of full test files. The input symbols are encoded using bytes for all files except **words**, for which we use 32 bits per symbol.

Input	$n/2^{30}$	$\lvert\Sigma\rvert$	$100r/n$	Σ_r/r
kernel	128.0	229	0.09	1494.76
geo	128.1	211	0.15	1221.49
wiki	128.7	213	16.71	29.40
dna	128.0	6	18.46	23.79
debruijn	128.0	2	99.26	35.01
words	12.5	97 002 175	42.49	5.17

Each of the two algorithms has thus four variants: depending on whether it uses parallelism, and whether it runs in the in-place mode. The implementations are capable of using arbitrary types to represent integers and text symbols, but for simplicity in all experiments in this section we use 40-bit integers. The implementations used in experiments (as well as datasets described next) are available at `http://www.cs.helsinki.fi/group/pads/`.

Setup. We performed experiments on a machine equipped with two six-core 1.9 GHz Intel Xeon E5-2420 CPUs (capable, via hyper-threading, of running 24 threads) with 15 MiB L3 cache and 120 GiB of DDR3 RAM. For experiments we limited the RAM in the system to 4 GiB (with the kernel boot flag) and all algorithms were allowed to use 3.5 GiB in all experiments. The machine had 6.8 TiB of free disk space striped with RAID0 across four identical local disks achieving a (combined) transfer rate of about 480 MiB/s (read/write).

The OS was Linux (Ubuntu 12.04, 64bit) running kernel 3.13.0. All programs were compiled using `g++` version 5.2.1 with `-O3 -march=native` options. All reported runtimes are wallclock (real) times. The machine had no other significant CPU tasks running and for all sequential algorithms only a single thread of execution was used for computation (we permit a constant number of extra threads as long as they do not perform computation, e.g., threads responsible for scheduling I/O requests are allowed). The parallel algorithms used the full parallelism available on the machine (24 threads), unless explicitly stated otherwise.

Datasets. For the experiments we used the following files (see Table 3 for some statistics):

- **kernel**: a concatenation of ∼10.7 million source files from over 300 versions of Linux kernel (see `http://www.kernel.org/`). This is an example of highly repetitive file;
- **geo**: a concatenation of all versions (edit history) of Wikipedia articles about all countries and 10 largest cities in the XML format. The resulting file is also highly repetitive;
- **wiki**: a concatenation of wikipedia, w-source, w-books, w-news, w-quote, w-versity, and w-voyage dumps dated 20160203 in XML (see `http://dumps.wikimedia.org/`);
- **dna**: a collection of DNA reads from multiple human genomes filtered from symbols other than $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$ and newline (see `http://www.1000genomes.org/`);
- **debruijn**: a binary De Bruijn sequence of order k is an artificial sequence of length $2^k + k - 1$ than contains all possible binary k-length substrings. It contains nearly n irreducible lcps (see [9, Lemma 5]) which is the worst case for EM-SI;
- **words**: a collection of natural language text parsed into words and converted into 4-byte integers, see `http://www.statmt.org/wmt16/translation-task.html`.

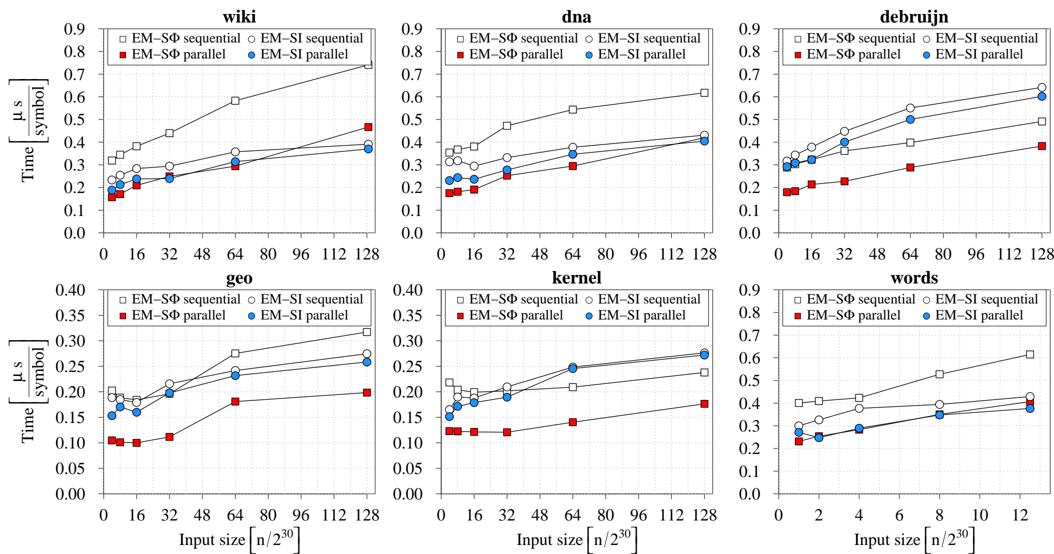


Figure 1 Scalability of the parallel algorithms compared to their sequential versions.

Parallel Algorithms. In the first experiment, we compare the parallel versions of the algorithms studied in this paper with their sequential versions. We executed the algorithms on increasing length prefixes of all testfiles and measured the runtime. For now we use all algorithms in the basic out-of-place mode. The results are given in Figure 1.

The parallel version of EM-SΦ achieves a significant speedup across all prefix sizes and input types. This is caused by the fact, that the algorithm performs a lot of random accesses to Φ_q in Step 1 and PLCP_q in Steps 2(a) and 2(c), making them strongly compute-bound. The parallelization helps even in Step 2(b), since most of the lcp comparisons are short which prevents OS from effective cache prefetching. The average speedup across all input types varies from 46% for 4 GiB inputs to 32% for full testfiles.

The speedup for EM-SI is notably smaller, since its sequential version is already largely I/O-bound. While for non-repetitive inputs the algorithm achieves a speedup of about 10%, for highly repetitive data (kernel, geo), the speedup is negligible, particularly for large text.

In the second experiment, we focus on the parallel version of EM-SΦ, as it benefits more from parallelism than EM-SI. We executed the algorithm on the largest instances of two testfiles (**wiki** and **geo** serving as examples of non-repetitive and highly-repetitive input) using different number of threads (we point out here, that the sequential version and parallel version running on a single thread are not the same implementation as the purely sequential version can avoid certain computations), and measured the runtime. As seen in Figure 2, the maximum speedup is already achieved with about 8 threads. At this point the algorithm becomes essentially I/O-bound.

In-Place Algorithms. In the next experiment we study the in-place variants of the algorithms described in Section 5. The in-place mode increases the I/O in both algorithms mostly due to additional scans of SA. The change in I/O volume is, however, not significant, hence for brevity we do not report I/O volume in this section. To study the effect on runtime, we executed the algorithms both in the in-place and the out-of-place mode on different prefixes of testfiles, and measured the runtime and disk space usage (to measure disk usage we used our own script but as a sanity check we ran a preliminary set of experiments in the in-place

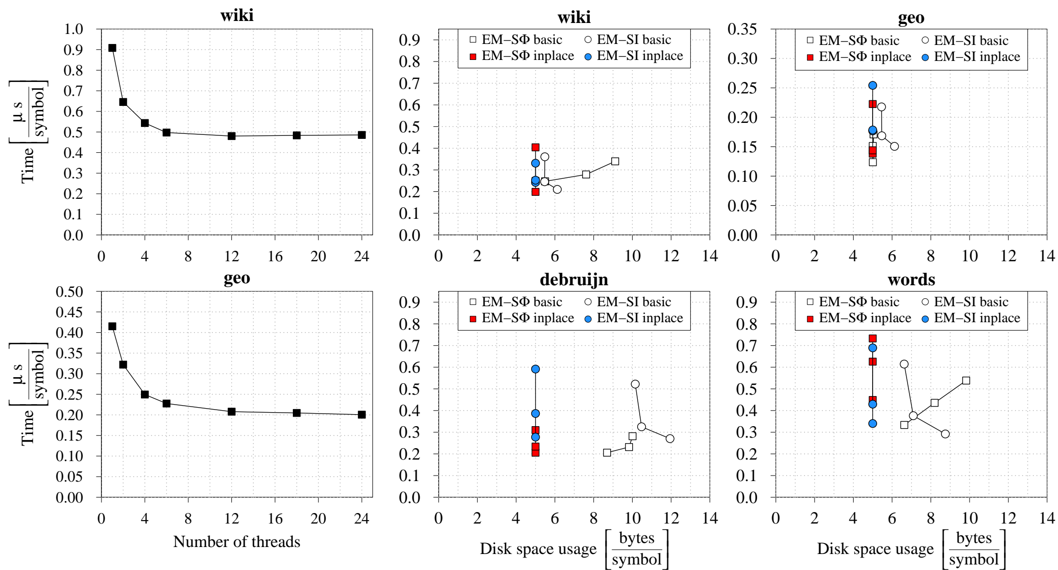


Figure 2 Left: Normalized runtime of the parallel version of EM-SΦ on the prefixes of length $n = 128 \times 2^{30}$. Right: Normalized runtime vs. disk space usage of the parallel algorithms in the in-place mode compared to the basic (out-of-place) mode. The three dots of each color/shape correspond to text prefixes of sizes 4, 16, and 64 GiB, which for the **words** file means $n \in \{2^{30}, 2^{32}, 2^{34}\}$.

Table 4 Comparison of two approaches to LCP array construction for large-alphabet inputs.

Algorithm	Time $\left[\frac{\mu s}{\text{symbol}}\right]$	Disk space $\left[\frac{\text{bytes}}{\text{symbol}}\right]$	I/O volume $\left[\frac{\text{bytes}}{\text{symbol}}\right]$
EM-SΦ native	0.41	8.43	137.25
EM-SΦ byte-based	1.00	34.07	254.12
EM-SI native	0.38	5.53	116.78
EM-SI byte-based	1.08	21.94	259.56

mode using a setup, where the available disk space is only negligibly larger than the output LCP array). For simplicity, we present the results for parallel versions but they were very similar on sequential versions. The results are given in Figure 2. We point out that due to the simultanous disk space measurement, these runs are slightly slower and thus not comparable to Figure 1, but the relative runtimes remain the same.

The slowdown of the in-place mode compared to basic versions is very moderate. The maximum slowdown for EM-SΦ was about 36% (but in most cases much smaller), and the maximum slowdown for EM-SI was about 17%. The working disk space usage in some cases, particularly for non-repetitive inputs, is reduced by more than a factor of two.

Large Alphabet. Suppose that the input string consists of symbols drawn from a large alphabet such that each symbol requires more than one byte. To compute the LCP array for such strings we can take one of two approaches. First, we can use an algorithm that natively supports large alphabet, and we have modified our implementations to provide such support. A recently published implementation of EM scalable and space-efficient suffix array construction provides a large-alphabet support, complementing this approach [8].

An alternative method is to first split each symbol into a group of symbols over byte alphabet. One can then apply a byte-based suffix sorter to compute the suffix array, then run a byte-based LCP array construction, and then compute and select the final subset of LCP

values in the postprocessing stage (which requires one scan of SA and LCP). A drawback of this approach is that reducing the alphabet increases the length of the string. For example, if the symbols of the original string of length n were encoded using 32-bit integers and we wish to obtain a string over byte alphabet, the resulting string has length $4n$.

To compare the two approaches in practice we used the parallel versions of the two algorithms studied in this paper in the basic (out-of-place) mode. We executed each algorithm on the prefix of the **words** testfile of length $n = 12.5 \times 2^{30}$ (with each symbol encoded using four bytes), first using the native large-alphabet mode, and then assuming the input is over byte alphabet, and compared the resources needed by the two approaches. We exclude the resources needed to compute the suffix array of the byte-interpreted input (needed in the second approach), as well as the postprocessing of the LCP array. The results, scaled with respect to the large-alphabet string, are reported in Table 4. Using the algorithm that natively supports large alphabet is about 2.5 × faster, uses half of the I/O volume, and requires about 4 × less working disk space. The overall disk usage (i.e., including input) of the native mode is even smaller, because the SA of the large-alphabet text is four times smaller than the SA of the byte-intepreted text.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.
2. T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and LCP arrays in external memory. In *Proceedings of the 15th Workshop on Algorithm Engineering and Experiments (ALENEX 2013)*, pages 88–102. SIAM, 2013. doi:10.1137/1.9781611972931.8.
3. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and Pat arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice–Hall, 1992.
4. J. Kärkkäinen and D. Kempa. Faster external memory LCP array construction. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57 of *LIPIcs*, pages 61:1–61:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.61.
5. J. Kärkkäinen and D. Kempa. LCP array construction in external memory. *J. Exp. Algorithmics*, 21(1):1.7:1–1.7:22, April 2016. doi:10.1145/2851491.
6. J. Kärkkäinen and D. Kempa. LCP array construction using O(sort(n)) (or less) I/Os. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, volume 9954 of *LNCS*, pages 204–217. Springer, 2016. doi:10.1007/978-3-319-46049-9_20.
7. J. Kärkkäinen, D. Kempa, and M. Piątkowski. Tighter bounds for the sum of irreducible LCP values. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 316–328. Springer, 2015. doi:10.1007/978-3-319-19929-0_27.
8. J. Kärkkäinen, D. Kempa, S. J. Puglisi, and B. Zhukova. Engineering external memory induced suffix sorting. In *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX 2017)*, pages 98–108. SIAM, 2017. doi:10.1137/1.9781611974768.8.
9. J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009. doi:10.1007/978-3-642-02441-2_17.

10 T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001. `doi:10.1007/3-540-48194-X_17`.

11 V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing.* Cambridge University Press, 2015.

12 U. Manber and G. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. `doi:10.1137/0222058`.

13 G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):article 2, 2007. `doi:10.1145/1216370.1216372`.

14 E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

15 K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 225–232. ACM/SIAM, 2002.

16 G. Tischler. Low space external memory construction of the succinct permuted longest common prefix array. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, volume 9954 of *LNCS*, pages 178–190. Springer, 2016. `doi:10.1007/978-3-319-46049-9_18`.

17 H. E. Williams and J. Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999. `doi:10.1093/comjnl/42.3.193`.

Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees*

Stefan Funke[1], Sören Laue[2], and Sabine Storandt[3]

1 **University of Stuttgart, Stuttgart, Germany**
funke@fmi.uni-stuttgart.de
2 **Friedrich-Schiller-Universität Jena, Jena, Germany**
soeren.laue@uni-jena.de
3 **JMU Würzburg, Würzburg, Germany**
storandt@informatik.uni-wuerzburg.de

Abstract

In a personalized route planning query, a user can specify how relevant different criteria as travel time, gas consumption, scenicness, etc. are for his individual definition of an optimal route. Recently developed acceleration schemes for personalized route planning, which rely on preprocessing, achieve a significant speed-up over the Dijkstra baseline for a small number of criteria. But for more than five criteria, either the preprocessing becomes too complicated or the query answering is slow. In this paper, we first present a new LP-based preprocessing technique which allows to deal with many criteria efficiently. In addition, we show how to further reduce query times for all known personalized route planning acceleration schemes by considering approximate queries. We design a data structure which allows not only to have personalized costs but also individual approximation guarantees per query, allowing to trade solution quality against query time at the user's discretion. This data structure is the first to enable a speed-up of more than 100 for ten criteria while accepting only 0.01% increased costs.

1998 ACM Subject Classification E.1 [Data Structures] Graphs and Networks

Keywords and phrases personalized route planning, contraction hierarchies, linear program, separation oracle, approximate queries

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.18

1 Introduction

While conventional route planning engines usually compute the shortest or quickest path between a given source and a target, individual preferences of users might differ. For example, a user might accept a slightly later arrival time at the target in exchange for a less crowded route, or reduced gas consumption, or fewer traffic lights or left turns on the way. To provide full flexibility, all possible trade-offs between all criteria should be valid definitions of an optimal route – and each query should be allowed its own definition. The personalized route planning problem captures this idea. Formally, it can be phrased as follows: Given a street network $G(V, E)$, with a d-dimensional non-negative cost vector $c(e) \in \mathbb{R}^d$ for each edge $e \in E$ (where each entry reflects one criterion, e.g. c_1 travel time, c_2 number of traffic lights, c_3 gas price, and so on); a query consists of source and target $s, t \in V$ and non-negative

* This work was supported by a Google Research Award on 'Personalized Route Planning'. Sören Laue acknowledges the support of Deutsche Forschungsgemeinschaft (DFG) under grant LA-2971/1-1.

© Stefan Funke, Sören Laue, and Sabine Storandt;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 18; pp. 18:1–18:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

weights $\alpha_1, \alpha_2, \cdots, \alpha_d$ (expressing the importance of each cost component for the user). The goal is to compute the path p from s to t in G which minimizes $\sum_{e \in p} \alpha^T c(e)$.

Dijkstra's algorithm which computes individual edge costs $\alpha^T c(e)$ on demand can solve the personalized route planning problem in $\mathcal{O}(n \log n + dm)$ with $|V| = n, |E| = m$. But for applications where efficiency is important – as in a server-client scenario with multiple queries per second – Dijkstra's algorithm does not allow for sufficient throughput. Therefore, like for the conventional route planning problem, preprocessing based speed-up techniques were investigated in previous work. But they either suffer from a complex preprocessing routine (based on contraction hierarchies [14]) which limits the practical applicability to $d \leq 3$ [13], [11], or with lighter preprocessing the speed-up for $d \geq 5$ is less than 100 compared to the Dijkstra baseline [10], [8], [12].

In this paper, we show how to instrument contraction hierarchies for the personalized route planning problem with the help of linear programming. Our method is conceptually simpler than previous attempts to use contraction hierarchies for this scenario. As a consequence, we can perform the preprocessing also for $d > 3$ which was impractical before. Furthermore, we describe a framework which allows to compute personalized queries with an approximation guarantee. Our respective data structure allows to trade query time against solution quality. Thereby the quality is a parameter to be set by the user, and can vary with every query. In our experiments, we show that accepting only slightly suboptimal results enables query answering two orders of magnitude faster than the Dijkstra baseline even for high-dimensional costs.

1.1 Related Work

The personalized route planning problem was introduced in [13]. There, a contraction hierarchy augmented with landmarks [15] was shown to provide a speed-up of three orders of magnitude over the Dijkstra baseline; but only for $d = 2$, and with some restrictions on the weights that can be chosen. In [11], a contraction hierarchy construction scheme was designed that maintains all optimal paths for arbitrary dimensions d and arbitrary weights α. But the CH construction relies on a rather complicated subroutine which involves the computation of a dynamically changing convex hull of points in $\mathbb{N}^d$. The speed-ups presented there were about three orders of magnitude for $d = 2$ and two orders of magnitude for $d = 3$. For larger d, no practical results could be provided.

To handle a larger number of metrics, the preprocessing paradigm for customizable route planning schemes was applied to personalized route planning [10], [8], [12]. Here the idea is to first construct an overlay graph independently of the metric(s). Then, in a second phase, the edges of the overlay graphs are augmented with suitable costs or cost vectors. In [10], the overlay graph was based on a k-path cover of the graph. This allowed to consider up to 64 metrics for the first time. But the speed-up was rather moderate: about 12 for $d = 2$ and 8 for $d = 64$. In [8], a slightly more general personalized route planning model was used, where also restrictions to binary weights and non-additive costs are possible. Here, the overlay graph is obtained by some simple rules which only rely on the topology of the graph. A speed-up of about 30 was reported for $d = 8$ and about 15 for $d = 64$. In [12], customizable contraction hierarchies [7] and customizable route planning [4] were turned into personalizable schemes. As the number of cost vectors of an edge in the overlay graph were shown to become huge, optimality-preserving pruning strategies were described. With that, the speed-up for $d = 2$ was reported to be about 100, for $d = 10$ about 70 and for $d = 64$ about 20.

Further speed-ups were achieved by making use of parallelization either utilizing multiple cores [6] or even the GPU [5]. Here, the whole graph is personalized for every query by setting all edge costs to $\alpha^T c(e)$. But in a client/server scenario, these approaches require too much machinery and space per user to allow for high throughput. Hence, in the remainder of the paper, we only consider sequential approaches.

In some way the dual problem of personalized routing was considered in [3] and [9]. Here, given a set of paths a user has traveled, the goal is to infer his weight vector α (also called his preference) that explains why exactly these paths were chosen. While in [3] a method based on stochastic coordinate descent was introduced, the approach in [9] relies on linear programming. We will use similar LP-based techniques as proposed in [9] to allow for efficient computation of personalized routes.

Suboptimal query answering for acceleration and more concise result representation was already considered in related domains such as public transit route planning. There, the multi-criteria setting is naturally induced, as besides travel time also the number of transfers, the walking time and the ticket prices (among other criteria) are relevant when planning a journey. As already with three criteria there can be hundreds or even thousands of Pareto-optimal route options between a source and a target, fuzzy dominance [2] and rule-based filters [1] were applied to reduce the solution set. But neither of those comes with a guarantee that assures the output in the end to be close to the optimal solution. In contrast, our method for approximate personalized routes will allow to fix an arbitrary approximation factor $\delta \geq 1$ and the returned solution will never be more than δ times as expensive as the optimal route.

1.2 Contribution

We provide the following new insights and algorithms for personalized route planning:

- For contraction hierarchies, the preprocessing in previous work [11] required a complicated binary-search like algorithm which involved the computation of a dynamically changing convex hull of d-dimensional points. We present a conceptually simpler, more efficient and robust LP-based contraction hierarchy (LP-CH). This approach also considerably reduces the theoretical runtime dependency on d for a crucial operation in the construction from $\mathcal{O}(d^2 \log(nMd)(n \log n + dm + d^{4+d} \log^{d-1}(nMd)))$ to $\mathcal{O}(d^2 \log(nMd)(n \log n + dm))$ where M is the maximum cost entry of any edge.
- We develop a new data structure which allows to answer personalized queries in an approximate fashion. Thereby, the approximation factor has not to be fixed a priori but can be chosen on query time. At the heart of our data structure lies an algorithm which sorts a list of d-dimensional cost vectors, such that the first i vectors approximate the complete set of vectors provably well for every i.
- We experimentally prove that our new contraction hierarchy preprocessing allows to deal with more than three metrics efficiently. Our framework for approximate query answering combines well with contraction hierarchies and also with approaches based on customizable route planning presented in previous work. This allows us to achieve further speed-ups when accepting slightly suboptimal results. For example, for $d = 10$ and an approximation factor of 1.001, query answering with our methods is two orders of magnitude faster than with bidirectional Dijkstra.

Comparisons between results from different papers are especially challenging for personalized route planning, as not only benchmark graphs might differ but more importantly also the criteria used. The selected criteria can affect the performance significantly: Well-correlated metrics as travel time and distance are easier to manage than conflicting objectives as travel

time and quietness (as illustrated already in [11]). We therefore make our largest benchmark instance with 10 meaningful metrics available[1] to alleviate future comparison of algorithms.

2 Contraction Hierarchies (CH) with an LP-Oracle

In this section, we describe how to construct contraction hierarchies for multi-dimensional costs. The main novelty compared to previous work is the usage of an LP-oracle in the preprocessing.

2.1 Conventional Contraction Hierarchies

The contraction hierarchies approach [13] computes an overlay graph in which so called shortcut edges span large sections of the shortest path. This reduces the hop length of optimal paths and therefore allows Dijkstra's algorithm to answer queries more efficiently.

The preprocessing is based on the so-called node contraction operation. Here, a node v as well as its adjacent edges are removed from the graph. In order not to affect shortest path distances between the remaining nodes, shortcut edges are inserted between all neighbors u, w of v, if and only if uvw was a shortest path (which can easily be checked via a Dijkstra run). The cost of the new shortcut edge (u, w) is set to the summed costs of (u, v) and (v, w). In the preprocessing phase all nodes are contracted one-by-one in some order. The rank of the node in this contraction order is also called the *level* of the node.

After having contracted all nodes, a new graph $G^+(V, E^+)$ is constructed, containing all original edges of G as well as all shortcuts that were inserted in the contraction process. An edge $e = (v, w)$ – original or shortcut – is called upwards, if the level of v is smaller than the level of w, and downwards otherwise. By construction, the following property holds: For every pair of nodes $s, t \in V$, there exists a shortest path in G^+, which first only consist of upwards edges, and then exclusively of downwards edges. This property allows to search for the optimal path with a bidirectional Dijkstra only considering upwards edges in the search starting at s, and only downwards edges in the reverse search starting in t. This reduces the search space significantly.

For personalized route planning, we are not dealing with scalar edge costs but cost vectors. Here, deciding the necessity of a shortcut in the CH graph is not trivial anymore. A shortcut (u, w) with costs $c(u, v) + c(v, w)$ has to be inserted upon contraction of v if and only if it encodes an optimal path p from u to w for some choice of α. But checking for all possible $\alpha \in [0, 1]^d$ whether p is the respective optimal path is obviously not possible. Therefore, we need a more efficient way to decide the necessity of a shortcut.

2.2 Shortcut Insertion Oracles

To keep the CH graph as sparse as possible, we only want to insert shortcuts which are necessary for exact query answering. So the only way to omit the insertion of a shortcut (u, w) which spans the edges (u, v) and (v, w) with costs $c(u, v), c(v, w)$ is to certify that there exists no α such that $\alpha^T c(u, v) + \alpha^T c(v, w)$ is the minimum cost among all paths p from u to w.

[1] https://www.dropbox.com/s/tclrjdkfhabu27h/ger10.zip?dl=0

Convex Hull Oracle. In [11], it was shown that the shortcut insertion problem is solvable in polynomial time for fixed dimension d by reduction to the following geometric problem: A path p with d-dimensional cost vector c can be interpreted as point in $\mathbb{N}^d$. It was shown that this path is optimal for some choice of α if and only if the respective point is part of the lower convex hull of all points that correspond to paths between the same source and target. Then an algorithm was described which decides if a point is part of the lower convex hull or not. As the convex hull might have exponential complexity, its explicit construction and inspection is impractical. The described algorithm avoids the explicit construction by iteratively computing optimal paths for α vectors chosen from a certain multi-dimensional interval. If the optimal path for some α equals p, the respective shortcut is necessary for sure and the search can be aborted. Otherwise, the optimal path p' allows to decrease the volume of the multi-dimensional interval of α vectors for which p could still be optimal by at least a constant fraction. Therefore, after at most polynomially (in the input size) many steps, the algorithm terminates and then certifies that no α exists for which p is optimal. The main problem with this approach is that calculating the new multi-dimensional interval for α and choosing a new reasonable α within this interval requires the computation of d-dimensional hyperplane cuts, which is expensive and numerically unstable. The total runtime of this approach was shown to be in $\mathcal{O}(d^2 \log(nMd)(n \log n + dm + d^{4+d} \log^{d-1}(nMd)))$ with $M = \max_{e \in E} |c(e)|_\infty$ being the maximum cost of any vector in any dimension.

We will now introduce an oracle based on linear programming which is simpler and leads to an improved theoretical runtime of $\mathcal{O}(d^2 \log(nMd)(n \log n + dm))$.

Naive LP Oracle. Our goal is still to either find an α for which our reference path p from u to w with costs $c(p) \in \mathbb{N}^d$ is optimal or to certify that no such α exists. We will achieve this by setting up an LP which has a feasible solution if and only if p is optimal for some α.

Let P be the set of all possible paths from u to w in G. We want to find an α such that $\alpha^T c(p)$ is not larger than $\alpha^T c(p')$ for all $p' \in P$, that is, p is optimal for this choice of α. We can express this as a simple system of linear (in)equalities or constraints:

$$\begin{aligned} \sum_{i=1}^{d} \alpha_i &= 1 && \\ \alpha_i &\geq 0 && i = 1, \dots, d \\ \alpha^T c(p) - \alpha^T c(p') &\leq 0 && \forall p' \in P \end{aligned}$$

Obviously, any α which is a solution for this linear program certifies that p is optimal for some queries. If there is no feasible solution, then for all choices of α there exists some path p' in P with lower costs and hence the shortcut encoding p can be omitted.

The problem with this LP is that – like for the convex hull oracle – the set P of alternative paths from u to w might be exponentially large. Hence setting up the LP is already too expensive to be practical.

Improved LP Oracle. The basic question is whether we really have to consider all alternative paths in P to decide whether the shortcut is necessary or not.

The ellipsoid method [16] for LP solving does not require all constraints to be explicitly available, but instead demands the existence of an efficient separation oracle. A separation oracle is fed with a possible solution and then either has to verify that this is indeed a solution for the complete LP (with all constraints) or otherwise has to return a (so far not explicit) constraint that is violated by the current solution.

In our setting, Dijkstra's algorithm can serve as separation oracle: For some choice of α, we can efficiently check whether the path p is optimal for this choice or not by running Dijkstra with edge costs $\alpha^T c(e)$. If p is not optimal, Dijkstra returns an alternative path p' which provides us with the new constraint $\alpha^T(c(p) - c(p')) < 0$. This constraint excludes all choices of α for which p' is a better route than p.

This gives rise to the following algorithm which returns true if p is optimal for some choice of α and false otherwise:

1. Initialize the LP with α with $\sum_{i=1}^{d} \alpha_i = 1$ and $\alpha_i \geq 0, i = 1, \ldots, d$.
2. Solve the LP using the ellipsoid method. If there is no solution, return false. Otherwise let the solution be α^*.
3. Run the Dijkstra separation oracle with α^*. If the optimal path is p, return true. If the optimal path is $p' \neq p$, add the constraint $\alpha^T(c(p) - c(p')) < 0$ to the LP. Go to 2.

The separation oracle runs in $\mathcal{O}(n \log n + dm)$. The number of iterations when using the ellipsoid method is bounded by $\mathcal{O}(k^2 L)$ where k is the number of variables and L the number of bits needed to represent the constraints. We have $k = d$ and as the coefficients represent costs of paths in G, they are bounded by nMd, hence we get $L = \log(nMd)$. In total, this results in a runtime of $\mathcal{O}(d^2 \log(nMd)(n \log n + dm))$.

2.3 Compacting Edges for Query Answering

As the result of the CH preprocessing, we get an overlay graph with shortcut edges. Each shortcut edge (u, w) is augmented with a d-dimensional cost vector, which is the result of aggregating the cost vectors along an optimal path from u to w. Naturally, there might be many such shortcuts between u and w, representing different optimal paths for different choices of α. During query answering, the relaxation of each of those edges might lead to a new temporary distance label for w, hence inducing multiple decrease key operations in the Dijkstra priority queue. If we instead merge all edges between u and w into a single edge – now with an associated set of cost vectors S – the relaxation of this edge consists of first computing $\min_{s \in S} \alpha^T s$, followed by at most a single decrease key operation, which is much more efficient. Furthermore, the data structure we get when using customizable route planning with personalization (as exploited in [10], [8], [12]) is also an overlay graph with sets of cost vectors per edge (but based on a different construction process).

From now on, we assume that some overlay graph for exact query answering is available. We can modify and augment any such overlay graph to enable efficient answering of approximate queries, as described in the next section.

3 Adaptive Approximation Guarantees

Among the set of potentially optimal routes between A and B, there are often many similar ones from a user perspective. For example, one route might have a travel time of 34 minutes and a gas price of 0.96 Euro, while another one has a travel time of 32 minutes and a gas price of 0.97 Euro. Furthermore, it is not easy for a user to specify α precisely in a meaningful way. Should the travel time be twice as important as the gas price, i.e. $\alpha^T = (2/3, 1/3)$, or three times as important, i.e. $\alpha^T = (3/4, 1/4)$? Slight variations in the choice of α might also result in different optimal routes. Hence we argue that two routes p and p' with very similar costs $\alpha^T c(p) \approx \alpha^T c(p')$ are almost indistinguishable for a user.

From the overlay construction, we get shortcut edges (v, w) with sets of cost vectors, encoding paths from v to w that are optimal for some choice of α. In particular for shortcuts

between nodes far away from each other, the set of cost vectors is typically quite large and hence relaxing the shortcut edge during query processing becomes very expensive (evaluating each vector with the given α). If one is willing to accept some small error, it might suffice just to inspect a few of these vectors. This gives rise to the following problem:

Consider a set of vectors $S = \{v \in \mathbb{R}^d\}$, $|S| = k$ and their objective function values $\alpha^T v$. We are interested in determining an ordering $v^{(1)}, v^{(2)}, \ldots v^{(k)}$ of the vectors in S as well as a respective sequence of error bounds $err^{(1)} \geq err^{(2)} \geq \cdots \geq err^{(k)} = 1$, such that for any optimization direction $\alpha = (\alpha_1, \ldots, \alpha_d)$, $\alpha_i \in \mathbb{R}_0^+$:

$$\frac{\min\limits_{j \leq i} \alpha^T v^{(j)}}{\min\limits_{v \in V} \alpha^T v} \leq err^{(i)}$$

that is, when only considering the first i vectors, we will never experience an objective function value worse than a factor $err^{(i)}$ than the optimum, no matter how the optimization direction α is chosen.

Clearly, any ordering of S will yield a monotonously decreasing sequence of error bounds, yet it is desirable to find an ordering where the errors drop as quickly as possible.

3.1 Two Greedy Strategies

In the following we present two greedy strategies to compute such an ordering. They both make use of a subroutine which for a given set of vectors $S' \subset S$ and an additional vector $w \notin S'$ computes the maximum relative approximation error of S' with respect to w, that is, how much worse the objective function value can get if S' is used to represent w. We will elaborate on this subroutine in the next subsection but treat it as a black box for now and denote by $T_{err}(k)$ the running time of this subroutine for a set of size k.

Iterative Error Minimization (bestNext). Assume we have already determined the first $i-1$ vectors $S^{(i-1)} := v^{(1)}, \ldots, v^{(i-1)}$ in this ordering. As next vector $v^{(i)}$ we want to choose the one in $S - S^{(i-1)}$ which minimizes the maximum relative error with respect to the remaining set. We can do so by iterating through all vectors $v \in S - S^{(i-1)}$ and computing the maximum relative approximation error of the set $S^{(i-1)} \cup \{v\}$ with respect to each $w \in S - (S^{(i-1)} \cup \{v\})$. As next $v^{(i)}$ we choose the candidate v minimizing this maximum relative approximation error. The error bound $err^{(i)}$ is set accordingly. For $|S| = k$, $O(k^2)$ calls to the black box subroutine are necessary to determine $v^{(i)}$. So in $O(k^3 T_{err}(k))$ time the ordering of S can be computed. Note that if for each $v \in S - S^{(i-1)}$ we remember its relative error wrt $S^{(i-1)}$, very often a call to the black box subroutine is not necessary if this 'old' relative error is below the worst relative error already found, effectively reducing the total running time to $O(k^2 T_{err}(k))$ in practice.

Worst-Error Next (worstNext). The number of calls to the black box subroutine might be prohibitive for very large sets S. The following alternative greedy strategy also produces an ordering using considerably less calls to the black box subroutine.

Again we assume that we have already determined the first $i-1$ vectors $S^{(i-1)} := v^{(1)}, v^{(2)}, \ldots v^{(i-1)}$ in this ordering. As next vector $v^{(i)}$ we choose a $v' \in S - S^{(i-1)}$ for which the relative approximation error of $S^{(i-1)}$ with respect to v' is maximized; $err^{(i)}$ is set accordingly. This requires $O(k)$ calls to the black box subroutine and hence $O(k^2 T_{err}(k))$ time to compute the ordering of S. As in the previous approach, calls to the black box

subroutine can often be saved if the 'old' relative errors are memorized, effectively reducing the running time in practice.

3.2 Bounding the Relative Approximation Error

Consider a set of vectors $S' = \{u^{(1)}, \ldots u^{(k)}\}$ and an additional vector $w \notin S'$. We are interested in the maximum *relative* approximation error of U with respect to w or more formally:

$$err_{rel} = \max_{\alpha} \frac{\min\limits_{u \in S'} u^T \alpha}{w^T \alpha} .$$

We will compute err_{rel} indirectly by determining the minimal factor $\delta \geq 1$ such that the vector δw is superfluous in (can be pruned from) the set $U \cup \{\delta w\}$ without affecting the optimum objective function value for any α. Since δ contributes linearly to the objective function, the minimal δ is exactly the relative approximation error err_{rel}, since then

$$\max_{\alpha} \frac{\min\limits_{u \in S'} u^T \alpha}{\delta w^T \alpha} = 1.$$

Determining whether a vector can be pruned from a set has been characterized in [12] as follows:

▶ **Lemma 1.** *A vector $v \in \mathbb{R}^d$ can be pruned from a set of vectors $S \subset \mathbb{R}^d$ if a convex combination v' of other vectors from S dominates v.*

Here domination refers to component-wise $\leq$.

Based on this Lemma and similarly to [12] we search for a convex combination u' of the k vectors in S' dominating the vector δw, minimizing δ.

$$\min \delta \tag{1}$$

$$\gamma_1 u^{(1)} + \gamma_2 u^{(2)} + \ldots \gamma_k u^{(k)} \leq \delta w \tag{2}$$

$$\sum \gamma_i = 1 \tag{3}$$

$$\gamma_i \geq 0 \tag{4}$$

Note that the $\leq$ in inequality (2) is to be understood component-wise in each of the d dimensions. Clearly, this LP has a feasible solution (just choose δ large enough). Since its dual is a linear program with $O(k)$ constraints in $O(d+1)$ variables, theoretically it can be solved in $O(k)$ time for fixed d using approaches like see [17]. In practice, state-of-the-art linear programming solvers suffice to quickly compute an optimum solution.

3.3 Query Answering with an Approximation Guarantee

We will use the introduced greedy strategies to sort the cost vectors for each shortcut and remember the approximation factor for each prefix. Then, in a query with desired approximation factor δ, we only consider the cost vectors on each shortcut until the respective approximation factor is at least as good as δ. In that way, it is guaranteed that the outputted path in the end has costs at most δc^* with c^* being the optimal costs. Note, that it does not make sense to apply the sorting schemes to shortcuts with only a few cost vectors, as the potential for saving is small but we introduce some space overhead by storing the approximation factors.

4 Experimental Results

We implemented the new CH preprocessing technique as well as the algorithms to obtain adaptive approximation guarantees in C++ (g++ 6.2.0). The edge costs as well as the weights c, α are represented as *doubles*. Performance was measured on a single core of an Intel Xeon CPU E3-1225v3 with 3.20GHz and 32GB RAM running Ubuntu Linux 16.10. For solving LPs we called the GLPK library in version 4.60.

4.1 Benchmark

We use the street network of Germany with 22,046,972 and 44,702,123 edges extracted from OSM[2] as a benchmark graph. As in [12], we constructed the following ten meaningful metrics: *distance* (euclidean distance with a precision of 1 meter), *travel time* (in seconds), *positive height difference* with the elevations of nodes in the road network being computed using SRTM data[3] (precision of one meter, 0 for downhill edges), *distance on large/medium/small roads* using OSM road categories as basis (allowing to penalize e.g. large streets which tend to be more crowded, or small village streets as they might be too narrow), *gas price* according to the formula in [13] (with one-tenth of a cent as basic unit), *energy consumption* for electric vehicles (in Watt) , *unit* (uniformly 1 per edge) allowing to distinguish between curvy and rather straight routes as in OSM curves are typically modeled by many small edges while long straight roads consist of few edges only, and *quietness* penalizing large roads and roads in dense road clusters (indicating city centers), with the penalty being proportional to the length of such a road, and zero for all others. We make the graph with all metrics available at `https://www.dropbox.com/s/tclrjdkfhabu27h/ger10.zip?dl=0`.

4.2 LP-based Contraction Hierarchy

We first investigate the performance of our new CH construction variant which uses linear programming with a Dijkstra-based oracle for efficiently deciding which shortcuts are necessary. We compare it to the previous version of CH which used a convex hull approach to decide the importance of a shortcut [11]; and to the CH variant based on customizable route planning where shortcuts are inserted independently of the metric and coated with cost vectors in a second preprocessing phase followed by some basic pruning, see [12]. We refer to these three variants as LP-CH (linear programming CH), H-CH (hull CH), and PC-CH (personalized customizable CH). For $d = 1$, we just use conventional CH.

4.2.1 Preprocessing

In Table 1, we provide the preprocessing time as well as the number of edges in the CH-graph and the number of cost vectors for all variants. We make the following observations: While H-CH is only applicable up to $d = 3$, LP-CH can also be computed for $d = 10$. The resulting graphs for H-CH and LP-CH are unsurprisingly similar. LP-CH is more efficiently computable for $d = 2$ as we use a simple oracle before actually starting the LP solver: We just compute the optimal path for all d possible α with a single 1 entry first. If the shortcut represents one of those paths, we add it to the graph. If one of those paths dominates the shortcut, we are sure the shortcut is unnecessary. Only if both cases do not apply, we start the LP-based

[2] `openstreetmap.org`
[3] `http://srtm.csi.cgiar.org/`

Table 1 Preprocessing results for real metrics. For $d = 2$, the process was stopped after 99.95% nodes were contracted, for $d = 3$ after 99.75% and for $d = 5, d = 10$ after 99% (to make PC-CH applicable). Timings are given in minutes, for # edges/vectors the 'm' denots millions. For comaprison, a CH for $d = 1$ can be constructed (with full contraction) in less than 10 minutes resulting a graph with 84.1 million edges.

	LP-CH			H-CH			PC-CH		
d	#edges	#vectors	time	#edges	#vectors	time	#edges	#vectors	time
2	85.6m	86.3m	25	85.6m	86.2m	46	94.5m	94.8m	24
3	84.1m	86.1m	32	84.2m	86.1m	23	89.3m	90.3m	28
5	75.2m	79.7m	28	-	-	-	84.4m	90.7m	47
10	76.4m	82.4m	56	-	-	-	84.4m	91.0m	152

algorithm described in Section 2.2. For $d = 2$, this reduces the number of LPs to solve significantly. Hence LP-CH is faster here as H-CH. For $d = 3$, more LPs have to be solved and the convex hull approach used for H-CH is slightly more efficient. In comparison to PC-CH, LP-CH leads to significantly smaller graphs. More advanced vector and edge pruning techniques could reduce the number of edges and vectors in the PC-CH graph further, but we observe that the preprocessing times are already worse for PC-CH for larger d. Also, further contraction of nodes for high values of d was completely impossible for PC-CH as the number of vectors grows too quickly. For LP-CH on the other hand, we could continue the contraction process up to 99.5% of the nodes. For $d = 5$ this took 32 minutes and resulted in a graph with 86.0 million vectors, for $d = 10$ it took about 2 hours and the final graph contained 92.6 million vectors. Note, that there might be some superfluous vectors contained in our LP-CH graphs as we allowed at most 100 LPs to be solved per shortcut in question, and just inserted the shortcut if we had no conclusive answer so far. Also, to accommodate for not having exact arithmetic on real numbers, we not only inserted a shortcut if it has exactly the optimal cost c^* for some α but also if the costs were less than $c^* + \epsilon$ with ϵ being an upper bound on the error that might occur.

4.2.2 Query Answering

Next, we compare LP-CH to PC-CH in terms of query processing efficiency. We measure speed-up towards the bi-directional Dijkstra in the original graph. Our results for varying d are summarized in Table 2. We observe that LP-CH outperforms PC-CH for all choices of d but the advantage decreases with higher dimension. While the PC-CH graph contains many more edges and vectors than the LP-CH graph, the query times for larger d are dominated by the relaxation of few shortcuts with large cost vector sets for both approaches. For $d = 5$, the LP-CH graph contains shortcuts with 455 vectors, for PC-CH up to 986 vectors. Figure 1 shows the distribution of the vector set sizes. Indeed, most of the shortcuts have very small vector sets. But the few with large sets are typically shortcuts between important nodes (contracted late in the preprocessing) and hence are contained in the search space of many long-distance queries. Therefore, if the the set sizes of such shortcuts can be reduced – as envisioned with our approximation framework – the query times can be further improved.

4.3 Adaptive Approximation via Vector Ordering

In Section 3 we have devised two strategies for ordering the vector set associated with a shortcut edge which are evaluated in the following.

As initial test data we used randomly generated (non-dominated) vector sets in 5 and 10 dimensions, as well as actual vector sets as constructed by our LP-CH; the results can be

Table 2 Query results (averaged over 100 source-target pairs) for randomly chosen α vectors. A poll refers to an extraction operation from the priority queue. For LP-CH and PC-Ch we list the improvements towards the Dijkstra baseline.

	Dijkstra		LP-CH		PC-CH	
d	time (ms)	polls	speed-up	poll ratio	speed-up	poll ratio
2	5,192	$1.4 \cdot 10^7$	425	616	112	286
3	5,617	$1.4 \cdot 10^7$	311	422	98	224
5	6,357	$1.5 \cdot 10^7$	118	259	87	185
10	7,126	$1.6 \cdot 10^7$	71	192	54	174

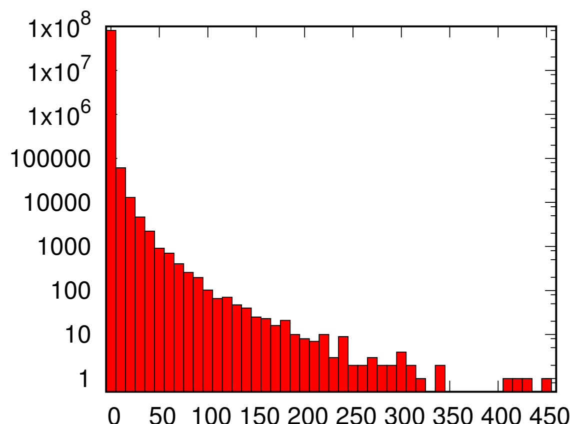


Figure 1 Number of shortcuts in dependency of the vector set size. The y-axis is in logscale.

seen in Table 3 (averages taken over 10 runs each). For example, inspecting only the first 32 vectors of a 10 dimensional vector set of size 130 guarantees on average an approximation ratio of 2.75 for the worstNext strategy and 2.90 for the bestNext strategy. Computing the former ordering took 23ms, the latter 873ms on average. As to be expected, the bestNext strategy takes considerably longer time, but sometimes yields better approximation guarantees when only considering very few vectors. We also see that random instances hardly mimic the instances actually appearing as vector sets of shortcuts due to the LP-CH construction. This is also comes as no surprise given the rather strong correlation between certain metrics in the real-world data. For the real-world instances, looking at the first 32 (out of 130) vectors yields almost perfect results (with an error below 1.0001). Just for comparison, using a *random order* leads to considerably worse approximation factors (e.g., 65 vs. 2.75 and 2.90 for the example mentioned above).

For our LP-CH we used the worstNext approach due to its quicker construction times compared to bestNext. We only considered shortcuts with 10 or more vectors for the ordering approach. There were less than $200,000$ such edges for all d. The results are given in Table 4. The additional preprocessing time is less than the time to compute the LP-CH in the first place. We see that already a very small set of vectors represent all vectors well, illustrated by the number of vectors necessary to guarantee an approximation factor of less than 1.001. We observe that in practice, the approximation factor is often even better, as parts of the path over shortcuts with small vector sets are exact and also the approximation factor computed by our approach assumes the worst possible α, which of course might not be the actual choice of the user.

We also observe that we indeed achieve larger speed-ups even for the very tight approximation factor of $\delta = 1.001$. This is more pronounced for larger dimensions d, where we have

Table 3 Performance of ordering for random and real-world vector sets of size $k = 130$ (averaged over 10 runs each).

	$d = 5$, random		$d = 10$, random		$d = 10$, real-world	
	worstNext	bestNext	worstNext	bestNext	worstNext	bestNext
err_2	4894.6100	918.6600	319.2000	159.8840	4.0977	1.8349
err_4	739.4000	141.3070	86.1920	54.0640	1.2210	1.1787
err_8	37.0480	28.3280	22.4890	21.1100	1.0338	1.0227
err_{16}	6.4786	7.5151	6.2003	7.0924	1.0042	1.0037
err_{32}	2.5864	2.9134	2.7582	2.9043	1.0000	1.0000
err_{64}	1.4722	1.4895	1.6170	1.6364	1.0000	1.0000
err_{128}	1.0094	1.0093	1.0184	1.0184	1.0000	1.0000
time	18ms	916ms	23ms	873ms	11ms	990ms

Table 4 Effect of vector reordering and approximate query answering. The ordering time (in minutes) is the total time taken by worstNext to reorder vectors and compute approximation guarantees for all shortcuts with at least 10 vectors. The avg. and max until good count the number of vectors after reordering until an approximation factor of 1.001 was achieved. The speed-up is then computed as average over 100 queries with $\delta = 1.001$.

d	ordering time	avg. until good	max until good	speed-up
2	< 1	4.1	15	478
3	5	6.5	20	357
5	14	8.4	22	176
10	23	12.1	47	131

larger cost sets per vector and hence save more by restricting us to looking at only a few of the vectors. So accepting only slightly suboptimal result, we achieve a speed-up of over 100 for $d = 10$ which was not possible before. We want to emphasize, that the approximation factor δ can be chosen *per query*, hence even faster query times can be achieved by relaxing δ, or more precise results at the cost of an increased running time.

5 Conclusions and Future Work

We introduced a new CH variant for the personalized route planning problem which allows to deal with more metrics than previous methods while exhibiting manageable preprocessing times and better speed-ups than approaches based on customization. But even with our new approach, the speed-up decreases considerably for a growing number of metrics, as large vector sets have to be inspected during query answering. As a remedy, we introduced a new scheme which allows to sort vectors in a way that the maximum relative error is guaranteed to be small when only few vectors are investigated. This scheme might be of independent interest and turn out to be useful in other scenarios with complicated edge costs or edge cost functions, as e.g. in time-dependent route planning. We experimentally proved that small approximation factors already allow to decrease the query times significantly.

In future work, it should be investigated if the reordering could be applied already during the preprocessing to accelerate it and maybe also allow for a later stop in the contraction process, which in turn might lead to better query times.

References

1. Hannah Bast, Mirko Brodesser, and Sabine Storandt. Result diversity for multi-modal route planning. In *ATMOS-13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 33, pages 123–136, 2013.
2. Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing multimodal journeys in practice. In *Proc. 12th International Symposium on Experimental Algorithms (SEA)*, pages 260–271. Springer, 2013.
3. Daniel Delling, Andrew V. Goldberg, Moises Goldszmidt, John Krumm, Kunal Talwar, and Renato F. Werneck. Navigation made personal: Inferring driving preferences from GPS traces. In *Proc. 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 31. ACM, 2015.
4. Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*, pages 376–387. Springer, 2011.
5. Daniel Delling, Moritz Kobitzsch, and Renato F. Werneck. Customizing driving directions with GPUs. In *Proc. 20th European Conference on Parallel Processing (EuroPar)*, pages 728–739. Springer, 2014.
6. Daniel Delling and Renato F. Werneck. Faster customization of road networks. In *Proc. 12th Int. Symposium on Experimental Algorithms (SEA)*, volume 13, pages 30–42. Springer, 2013.
7. Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In *Proc. 13th Int. Symposium on Experimental Algorithms (SEA)*, pages 271–282, 2014.
8. Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Fast exact shortest path and distance queries on road networks with parametrized costs. In *Proc. 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 66. ACM, 2015.
9. Stefan Funke, Sören Laue, and Sabine Storandt. Deducing individual driving preferences for user-aware navigation. In *Proc. 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 14. ACM, 2016.
10. Stefan Funke, André Nusser, and Sabine Storandt. On k-path covers and their applications. *Proceedings of the VLDB Endowment*, 7(10), 2014.
11. Stefan Funke and Sabine Storandt. Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In *Proc. 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013*, pages 41–54, 2013.
12. Stefan Funke and Sabine Storandt. Personalized route planning in road networks. In *Proc. 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 45. ACM, 2015.
13. Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route planning with flexible objective functions. In *Proc. 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 124–137, 2010.
14. Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.
15. Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete algorithms*, SODA'05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.
16. Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.
17. Raimund Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. Symp. on Computational Geometry (SCG)*, pages 211–215, New York, NY, USA, 1990. ACM.

Consumption Profiles in Route Planning for Electric Vehicles: Theory and Applications*

Moritz Baum[1], Jonas Sauer[2], Dorothea Wagner[3], and Tobias Zündorf[4]

1 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`moritz.baum@kit.edu`

2 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`jonas.sauer@student.kit.edu`

3 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`dorothea.wagner@kit.edu`

4 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`zuendorf@kit.edu`

Abstract

In route planning for electric vehicles (EVs), *consumption profiles* are a functional representation of optimal energy consumption between two locations, subject to initial *state of charge (SoC)*. Efficient computation of profiles is a relevant problem on its own, but also a fundamental ingredient to many route planning approaches for EVs. In this work, we show that the complexity of a profile is at most linear in the graph size. Based on this insight, we derive a *polynomial-time* algorithm for the problem of finding an energy-optimal path between two locations that allows stops at charging stations. Exploiting efficient *profile search*, our approach also allows *partial recharging* at charging stations to save energy. In a sense, our results close the gap between efficient techniques for energy-optimal routes (based on simpler models) and $\mathcal{NP}$-hard time-constrained problems involving charging stops for EVs. We propose a practical implementation, which we carefully integrate with Contraction Hierarchies (CH) and A* search. Even though the practical variant formally drops correctness, a comprehensive experimental study on a realistic, large-scale road network reveals that it *always* finds the optimal solution in our tests and computes even long-distance routes with charging stops in less than 300 ms.

1998 ACM Subject Classification G.2.2 Graph Theory, G.2.3 Applications

Keywords and phrases electric vehicles, charging station, shortest paths, route planning, profile search, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.19

1 Introduction

Route planning services explicitly designed for EVs have to address specific aspects, since EVs usually employ a rather limited cruising range. We study the problem of computing routes that minimize energy consumption, in order to maximize cruising range and for drivers to overcome *range anxiety* (the fear of getting stranded). This imposes nontrivial challenges. *Recharging* en route may become inevitable on long-distance trips. Given that charging stations are scarce, such stops need to be planned in advance [7, 42]. Moreover, EVs can *recuperate* energy (e. g., when going downhill), but the *battery capacity* limits the amount

* Tobias Zündorf's research was supported by DFG Research Grant WA 654/23-1.

© Moritz Baum, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 19; pp. 19:1–19:18

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of recuperable energy [9, 18, 36]. As a result, the energy-optimal route depends on the initial SoC. This dependency is captured by the notion of *(consumption) profiles*, which map SoC (at the source) to (minimum) energy consumption that is necessary to reach the target [9, 18, 38]. Profiles are relevant in many applications, where the SoC at the start of a journey is either unknown or can be decided by the driver, e. g., when charging overnight. Moreover, they are an important ingredient of *speedup techniques*, where preprocessing is applied to the input network for faster query times [7, 9, 18].

In this work, we examine the complexity of consumption profiles in road networks. Furthermore, we discuss an important application that requires efficient computation of profiles, namely, energy-optimal routes with intermediate charging stops. Even when optimizing for energy consumption only, the integration of charging stations into route planning is a nontrivial task: Recharging to a full battery at a charging station can be wasteful if it prevents the battery from recuperating energy on a downhill ride later on. Hence, profiles help in deciding the optimal amount of energy to be recharged at a station.

Related Work. Classic route planning approaches apply Dijkstra's algorithm [17] to a graph representation of the network, with fixed scalar edge costs representing, e. g., travel time. For faster running times in practice, *speedup techniques* [4] accelerate online shortest-path queries with data *preprocessed* in an offline phase. Examples of such techniques are CH [20], where vertices are contracted iteratively and replaced by *shortcuts* in the graph, and variants of A* search [21, 24]. Combining both techniques, Core-ALT [6] contracts most vertices and runs A* on the remaining *core graph*. Some techniques were also extended to more complex scenarios, such as time-dependent cost functions [5, 10, 12, 14]. In this context, a *profile query* asks for a functional representation of travel time between locations for *any* departure time. Such functions may have superpolynomial complexity [19], but can be computed by an output-sensitive search algorithm [11, 15]. See Bast et al. [4] for a more complete survey.

Regarding route planning for EVs, computing routes that minimize energy consumption requires the integration of battery constraints into Dijkstra's algorithm and adaptation of speedup techniques for fast queries [9, 18, 36]. Eisner et al. [18] observe that consumption profiles have *constant* complexity for a fixed *path*. Subsequent works consider *general* profiles possibly comprising *multiple* paths [9, 38], but their complexity has not been studied. Stops at charging stations are often considered under the simplifying assumptions that the charging always results in a *fully* recharged battery [22, 32, 40, 41, 42, 43]. Routes with a minimum number of intermediate charging stops can then be computed in less than a second on subcountry-scale graphs [41, 42]. More complex models also consider constraints on time spent driving [8, 30] and recharging [7, 27, 31, 33, 44, 45], but this results in much more difficult (typically $\mathcal{NP}$-hard) problems and proposed techniques are inexact or impractical.

Contribution and Outline. In Section 2, we formally introduce our model, describe problem variants, and recap basic algorithmic ingredients needed to solve them. In Section 3, we investigate the complexity of consumption profiles. As our main result, we prove that such profiles have linear complexity—much in contrast to profiles in time-dependent routing, which can have superpolynomial size [19]. This enables us to compute *profile search* efficiently—an algorithm relevant in various query scenarios and a crucial ingredient for many speedup techniques. In Section 4, we consider energy-optimal routes that allow stops at charging stations to recharge the battery. Unlike previous studies [22, 40, 41], we do not assume that using a charging station always results in a fully recharged battery. Instead, we allow the charging process to be *interrupted* beforehand to save energy. Building upon our theoretical

findings, we derive a *polynomial-time* algorithm to solve the problem. To make the approach fast in practice, we propose a (heuristic) variant and integrate it with CH [18, 20] and A* search [24]. Section 5 presents our experimental study, in which we demonstrate that our algorithm (empirically) obtains *optimal* results for all queries in well below a second after moderate preprocessing effort. We conclude with final remarks in Section 6.

2 Model, Query Variants, and Basic Algorithms

We model the road network as a directed graph $G = (V, E)$. Energy consumption along edges is given by the cost function $c\colon E \to \mathbb{R}$. Consumption can be negative to model recuperation, but cycles with negative consumption are physically ruled out. An EV is equipped with a battery of limited capacity $M \in \mathbb{R}_{\geq 0}$. Given the current SoC $b_u \in [0, M]$ of a vehicle at a vertex $u \in V$, traversing an edge $(u, v) \in E$ typically results in the SoC $b_v = b_u - c(u, v)$ at v. However, we also take *battery constraints* into account: The SoC b_v must neither exceed the limit M, nor drop below zero [1, 2, 18]. Thus, if the consumption $c(u, v)$ of an edge (u, v) exceeds the SoC b_u at u, the edge cannot be traversed, as the battery would run empty along the way. We indicate this case by setting $b_v := -\infty$. Conversely, if the battery is (almost) fully charged, passing an edge with negative consumption cannot increase the SoC beyond the maximum value M, so we obtain $b_v \leq M$. Given some initial SoC b_s at the source vertex s, we say that an s–t path $P = [s, \ldots, t]$ in G is *feasible* if and only if the battery never runs empty, i. e., the SoC b_v obtained at every vertex v of P after iteratively applying above constraints is within the interval $[0, M]$. Note that a path may be infeasible even if its *cost* (i. e., the sum of its consumption values) does not exceed b_s: Due to negative edge weights, there might be a prefix of greater total cost that renders the path infeasible. Given the SoC b_t at the target t of a feasible path, the *energy consumption* on the path is $b_s - b_t$. This value can become negative, due to recuperation.

For the sake of simplicity and without loss of generality, we assume in this work that $c(e) \in [-M, M]$ for all edges of the input graph. Moreover, we assume that shortest paths (wrt. the cost function c) between arbitrary pairs of vertices are unique.

Profiles. Given two vertices $s \in V$ and $t \in V$ of the input graph G, we define the *SoC function* $f\colon [0, M] \cup \{-\infty\} \to [0, M] \cup \{-\infty\}$, also called *SoC profile* or *s–t profile*, which maps SoC at the source s to the maximum SoC at the target t (after traversing any s–t path in G). We define $f(-\infty) := -\infty$. SoC functions are *piecewise linear*, so we use a sequence $F = [(x_1, y_1), \ldots, (x_k, y_k)]$ of *breakpoints* to define f, such that $f(b) = -\infty$ for $b < x_1$, $f(b) = y_k$ for $b \geq x_k$, and $f(b)$ is obtained by linear interpolation in all other cases.

For some s–t path P, we denote by f_P the profile of P, i. e., the SoC function that maps initial SoC at s to the resulting SoC at t after traversing P. Given the SoC functions f_P and f_Q of two paths P and Q, we say that f_P *dominates* f_Q (similarly, P *dominates* Q) if $f_P(b) \geq f_Q(b)$ for all $b \in [0, M]$.

To give a simple example, consider the SoC function $f_{[u,v]}$ induced by an edge $(u, v) \in E$. Given the cost $c(u, v)$ of the edge, battery constraints yield

$$f_{[u,v]}(b) := \begin{cases} -\infty & \text{if } b - c(u, v) < 0, \\ M & \text{if } b - c(u, v) > M, \\ b - c(u, v) & \text{otherwise.} \end{cases}$$

The function $f_{[u,v]}$ is represented by the sequence $F_{[u,v]} = [(c(u, v), 0), (M, M - c(u, v))]$ if

$c(u,v) \geq 0$ is nonnegative, and $F_{[u,v]} = [(0, -c(u,v)), (M + c(u,v), M)]$ otherwise. In both cases, the function consists of two breakpoints.

Query Variants. Typically, there are two problem variants of interest. First, an *SoC query* consists of a source $s \in V$, a target $t \in V$, and an initial SoC $b_s \in [0, M]$. It asks for a (single) *energy-optimal s–t* path when departing at s with SoC b_s, i. e., a path that maximizes the SoC b_t at t (and minimizes the consumption $b_s - b_t$). Second, a *profile query* does not take b_s as input, but asks for an s–t *profile*, i. e., the optimal value b_t for *every* initial SoC $b_s \in [0, M]$.

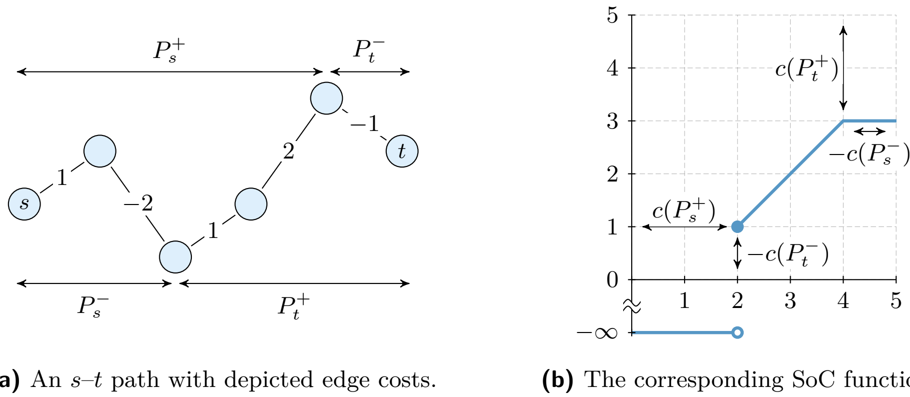
Basic Algorithms. SoC queries can be answered by a variant [1, 2] of Dijkstra's algorithm that handles battery constraints on-the-fly, using explicit checks. However, the algorithm is *label correcting*, i. e., negative edge costs may cause (exponentially many) re-scans of vertices [28]. *Potential shifting* [29] can remedy this, making use of a potential function $\pi \colon V \to \mathbb{R}$ on the vertices with the property that $c(u,v) - \pi(u) + \pi(v) \geq 0$ holds for all $(u,v) \in E$. Then the search becomes *label setting* when using the *reduced* cost function $c(u,v) - \pi(u) + \pi(v)$, i. e., each vertex is scanned at most once. Distances wrt. the original cost function are maintained and battery constraints can still be applied on-the-fly [18, 36]. We refer to this algorithm as *EV Dijkstra (EVD)*.

Profile search [9, 37, 38] is a label-correcting algorithm that answers profile queries. Starting from the source vertex $s \in V$, the algorithm maintains, for each vertex v, a (tentative) s–v profile f_v. It initializes $f_v \equiv -\infty$ for all $v \in V$, except $f_s = \mathrm{id}$. It adds s to a priority queue, which uses the value $\min_{b \in [0,M]}\{b - f_v(b)\}$ as key of a vertex v, i. e., the *minimum energy consumption* of its SoC function. In each step of the main loop, the algorithm extracts a vertex u with minimum key from the queue (thereby *scanning* it) and proceeds along the lines of Dijkstra's algorithm: Incident outgoing edges $e = (u,v)$ are *scanned* by computing the composition $f = f_e \circ f_u$. Afterwards, it sets $f_v = \max(f_v, f)$, i. e., the pointwise maximum of f_v and f, and updates v in the priority queue, if its key changed. After termination, the label f_v of every vertex $v \in V$ is the s–v profile.

3 On the Complexity of Profiles

In this section, we first examine characteristics of SoC functions of single paths. Then, we show that the complexity (i. e., number of breakpoints) of *general* SoC functions is linear in the number $|V|$ of vertices in the graph. An alternative notion that is common in the literature [7, 8, 9, 18] utilizes *consumption profiles* $g(\cdot)$, which map initial SoC to *energy consumption* between two vertices. We obtain the relation $f(b) = b - g(b)$, hence our insights on the complexity of SoC profiles carry over to consumption profiles directly.

Profiles Representing Paths. Given an s–t path P, Eisner et al. [18] show that the number of breakpoints of the SoC function f_P is bounded by a constant. Below, Lemma 1 recaps this fundamental insight, but also provides a full specification of the SoC function of a single path based on the costs of certain *subpaths*. We begin by defining *important* subpaths of an s–t path P. First, let P_s^+ denote the *maximum prefix* of P, i. e., the prefix of P that has maximum cost $c(P_s^+)$ wrt. energy consumption among all its prefixes. (Recall that the cost of a path is defined as the sum of its edge costs, hence, battery constraints do not apply.) If every prefix of P (including P itself) is negative, we obtain $P_s^+ = [s]$ and $c(P_s^+) = 0$. Similarly, the *minimum prefix* P_s^- minimizes the cost $c(P_s^-)$ among all prefixes of P. We obtain $P_s^- = [s]$ and $c(P_s^-) = 0$ in case every prefix of P is positive. The *maximum suffix*



(a) An s–t path with depicted edge costs. **(b)** The corresponding SoC function.

Figure 1 An s–t path together with its SoC function, assuming that the battery capacity is $M = 5$. The cost of the path is 1 and its important subpaths are indicated. Relative vertical positions of vertices correspond to costs of subpaths starting or ending at the respective vertex. The coordinates of breakpoints in the profile are equal to the costs of certain important subpaths.

P_t^+ and *minimum suffix* P_t^- are defined symmetrically. For the sake of simplicity, we assume that P contains no subpath with cost 0 consisting of more than one vertex (this can be enforced by perturbation of edge costs). Thus, the subpaths above are uniquely defined. Moreover, observe that $P = P_s^- \circ P_t^+ = P_s^+ \circ P_t^-$; see Figure 1.

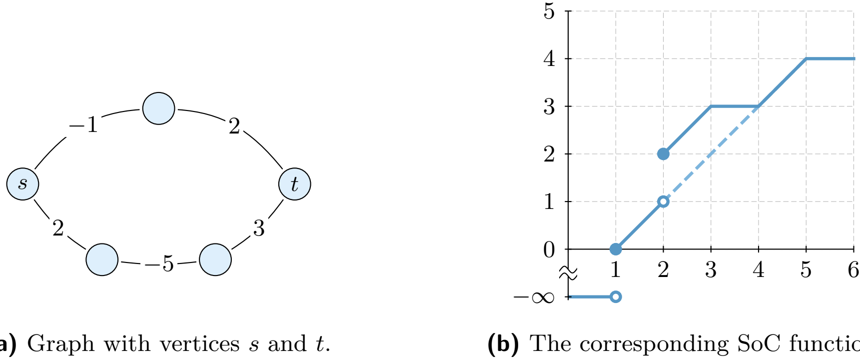
Lemma 1 shows that the SoC function f_P (defined by its breakpoints) of a path P is completely determined by the costs of its important subpaths. At most two breakpoints are necessary to represent the SoC function. It has a characteristic form: It consists of a first part with infinite consumption (the path is infeasible for low SoC), followed by a segment with slope 1 (the consumption is constant, thus SoC at t increases with SoC at s), and a last segment of constant SoC (for high values of initial SoC, the battery is fully charged at some point due to recuperation). Each of these three intervals may collapse to a single point. The segment with slope 1 is also called the *characteristic* segment of the SoC function. An example of a path and its SoC function is depicted in Figure 1.

▶ **Lemma 1.** *Given an s–t path P, its SoC function f_P is a piecewise linear function. It is defined by a sequence F of breakpoints in the following way.*
1. *If there exists a subpath of P with cost greater than M, $F = \emptyset$ and $f_P \equiv -\infty$.*
2. *Otherwise, if there is a subpath of P with cost below $-M$, $F = [(c(P_s^+), M - c(P_t^+))]$.*
3. *If neither such subpath exists, $F = [(c(P_s^+), -c(P_t^-)), (M + c(P_s^-), M - c(P_t^+))]$.*

General Profiles. For a pair of vertices s and t, different paths may be the optimal choice for different values of initial SoC; see Figure 2. Therefore, a general SoC function is the upper envelope of a set of SoC functions, each corresponding to a single path. We say that an s–t path *contributes* to the s–t profile if it is optimal for some initial SoC. We bound the number of breakpoints in the SoC function subject to the number of contributing paths. The following Lemma 2 is a direct implication of the observations by Atallah [3]. (In general, the number of breakpoints in the upper envelope of linear functions can be superlinear [46].)

▶ **Lemma 2.** *Given the set $\mathcal{P}$ of all contributing paths of an s–t profile, the number of breakpoints in the SoC function is linear in $|\mathcal{P}|$.*

Since the number of s–t paths can be exponential in the graph size, Lemma 2 does not yield an immediate polynomial bound on the complexity of the s–t profile. Note that in the



(a) Graph with vertices s and t. **(b)** The corresponding SoC function.

Figure 2 The SoC profile of given vertices s and t. The battery capacity is $M = 6$. The dashed segment indicates dominated parts of the SoC function of the upper s–t path.

related scenario of time-dependent profiles, it was shown that the number of contributing paths can actually become superpolynomial in the graph size [19]. In contrast, we now show that the number of breakpoints of an SoC function is in fact *linear* in the number of vertices of the input graph in the worst case.

We start with basic properties of paths and their SoC functions. Lemma 3 claims that a path P dominates another path Q if it is shorter (wrt. the cost function c) and both its maximum prefix and maximum suffix are shorter than the respective subpaths of Q. This follows immediately from the structure of SoC functions and is illustrated in Figure 3.

▶ **Lemma 3.** *Let P and Q be two s–t paths, such that $c(P_s^+) \leq c(Q_s^+)$, $c(P_t^+) \leq c(Q_t^+)$, and $c(P) \leq c(Q)$. Then the SoC function f_P of P dominates the SoC function f_Q of Q.*

As argued above, certain subpaths of an s–t path P are relevant to determine the corresponding profile. We add the following definitions. The *bottom vertex* v^- is the last vertex of the minimum prefix (and the first vertex of the maximum suffix) of P. Similarly, the *top vertex* v^+ denotes the last vertex of the maximum prefix (and the first vertex of the minimum suffix) of P. We call v^- and v^+ the *important* vertices of P. We presume that $v^- \neq v^+$, which always holds except in the trivial case $s = t$. The important vertices then separate P into three subpaths. (In case s or t are important vertices, one or two of these subpaths may consist of a single vertex.) Moreover, we distinguish two *types* of paths, depending on the order of appearance of important vertices. A path is called *top-bottom path* if v^+ appears before v^-, otherwise it is a *bottom-top path.* Lemma 4 states that prefixes and suffixes of contributing paths are uniquely defined by their corresponding important vertices.

▶ **Lemma 4.** *Given two vertices $s \in V$ and $t \in V$, let $v \in V$ be an arbitrary fixed vertex. All paths of the same type contributing to the s–t profile with v as their first important vertex share the same s–v subpath. Moreover, all contributing paths of the same type with v as their second important vertex share the same v–t subpath.*

Proof. Assume for contradiction that there are two contributing paths P and Q of the same type, such that the first important vertex of each path is v, but their respective s–v subpaths differ. Without loss of generality, let the s–v subpath of P be shorter. We replace the s–v subpath of Q by the subpath of P, which yields a modified path Q'. Clearly, the total length of Q' is below the length of Q, i. e., $c(Q') < c(Q)$. At the same time, neither the maximum prefix nor the maximum suffix of Q' exceeds the cost of the respective subpath

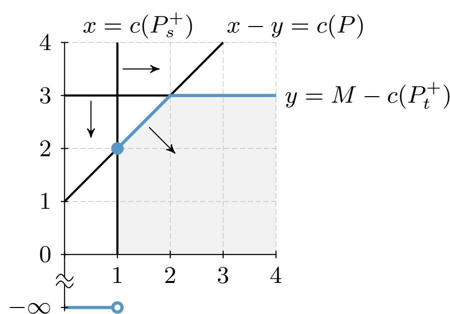


Figure 3 Dominated area of an SoC function, for a path P with $c(P) = -1$. Its maximum prefix and maximum suffix have cost $c(P_s^+) = c(P_t^+) = 1$. The costs induce three lines, each of which subdivides the Euclidean plane into two half planes. The SoC function of a path Q with $c(Q) \geq c(P)$, $c(Q_s^+) \geq c(P_s^+)$, and $c(Q_t^+) \geq c(P_t^+)$ lies in the shaded intersection of three of these half planes.

of Q. By Lemma 3, the modified path Q' dominates Q, contradicting the assumption that Q is a contributing path.

Similarly, we can replace the v–t subpath in one of two paths of the same type that share the second important vertex v by a shorter v–t subpath. Again, we obtain a new path that is shorter, while the lengths of its maximum prefix and suffix do not increase. Hence, at least one of the two paths does not contribute to the profile. ◀

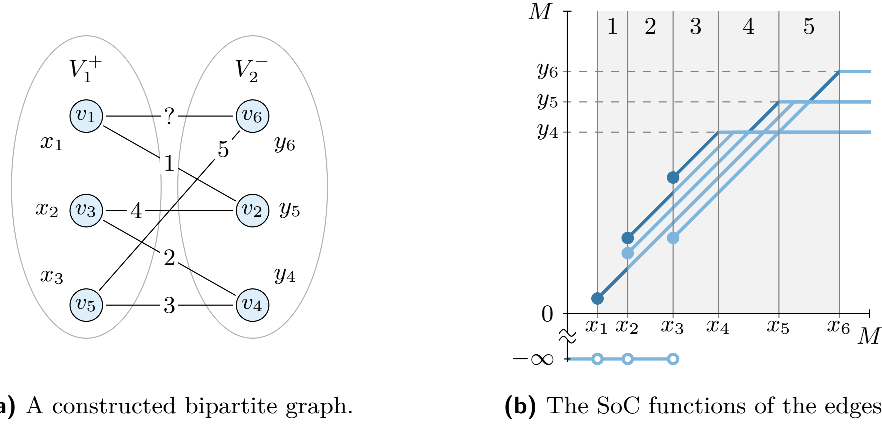
Below, Lemma 5 shows that together with their order in the path, pairs of important vertices uniquely define contributing paths of the same type. Note that this already implies that there are at most $\mathcal{O}(|V|^2)$ paths contributing to an s–t profile. Afterwards, we use a somewhat more sophisticated argument to show that the number of breakpoints is at most linear in the number of vertices.

▶ **Lemma 5.** *Let $s \in V$, $t \in V$, $v^- \in V$, and $v^+ \in V$ be four vertices of the input graph. There is at most one bottom-top path contributing to the s–t profile that has v^- as its bottom vertex and v^+ as its top vertex. Similarly, at most one contributing top-bottom path has v^+ as its top vertex and v^- as its bottom vertex.*

Proof. Assume for contradiction that there exist two distinct contributing s–t paths P and Q, such that both are bottom-top paths, their bottom vertex is v^-, and their top vertex is v^+. By Lemma 4, we know that P and Q share the same s–v^- subpath and the same v^+–t path. Hence, their v^-–v^+ subpaths must differ. Without loss of generality, let the v^-–v^+ subpath of P be shorter. Apparently, the total cost of the path P is lower than the cost of Q, i. e., $c(P) < c(Q)$. Similarly, the cost of the maximum prefix (suffix) of P is at most the cost of the maximum prefix (suffix) of Q. By Lemma 3, this implies that P dominates Q, contradicting the fact that Q contributes to the optimal solution. The other case is symmetric, so the claim follows. ◀

Using a somewhat more sophisticated argument, Theorem 6 shows that the number of breakpoints is at most linear in the number of vertices. It is easy to construct an example where the number of breakpoints in the SoC function is in fact linear in $|V|$, so this bound is tight up to a constant factor. This also enables profile search to run in $\mathcal{O}(|V|^2 \log |V|)$ time.

▶ **Theorem 6.** *Given a source $s \in V$ and a target $t \in V$ in the input graph $G = (V, E)$, the number of contributing paths (and breakpoints) in the s–t profile is in $\mathcal{O}(|V|)$.*



(a) A constructed bipartite graph.

(b) The SoC functions of the edges.

Figure 4 Illustration of the proof of Theorem 6. The constructed bipartite graph G' contains copies of top and bottom vertices of the input graph. Edges represent paths connecting certain important vertices. Vertices have assigned constants $x_1, x_2, x_3, y_4, y_5, y_6$. Edge labels indicate intervals depicted in Figure 4b, where the corresponding characteristic segments are contained in the upper envelope. Characteristic segments connect vertical lines induced by constants $x_1, \dots, x_6$. Parts of characteristic segments that lie on the upper envelope are highlighted (dark blue). Adding the missing characteristic segment that connects the lines induced by x_1 and x_6 (to form a cycle in the graph) results in at least one dominated SoC function.

Proof. To prove the claim, we construct a graph G' consisting of vertices representing important vertices in G. Edges of G' represent contributing paths between pairs of important vertices. We examine the structure of SoC functions of contributing paths to show that the number of edges in this graph is in $\mathcal{O}(|V|)$.

We construct an undirected graph G' that consists of the union of four sets of vertices $V_1^- = \{v_1^- \mid v \in V\}$, $V_1^+ = \{v_1^+ \mid v \in V\}$, $V_2^- = \{v_2^- \mid v \in V\}$, and $V_2^+ = \{v_2^+ \mid v \in V\}$. Clearly, the number of vertices in G' is linear in the number of vertices in the original graph. We add one undirected edge for every s–t path in the original graph that contributes to the SoC function: For every contributing bottom-top path with first important vertex u and second important vertex w, we add the edge $\{u_1^-, w_2^+\}$. For every contributing top-bottom path with first important vertex u and second important vertex w, we add the edge $\{u_1^+, w_2^-\}$. Lemma 5 implies that there are no multi-edges in the resulting graph. By construction, G' consists of at least two components and each component induces a bipartite subgraph. We claim that G' contains no cycles. This implies that the resulting graph has at most $\mathcal{O}(|V|)$ edges, which proves the theorem.

Assume for contradiction that there is a cycle $C = [v_1, \dots, v_k, v_1]$ in the graph constructed above. There are two possible cases: Either all edges in the cycle correspond to top-bottom paths and it contains only vertices in $V_1^+ \cup V_2^-$, or all edges correspond to bottom-top paths and all its vertices are in the set $V_1^- \cup V_2^+$.

Case 1. All edges represent top-bottom paths, and therefore $\{v_1, \dots, v_k\} \subseteq V_1^+ \cup V_2^-$. Figure 4a shows an example. Consider the profile induced by all paths corresponding to the edges of this cycle. Edges incident to some vertex $v_i \in V_1^+$, with $i \in \{1, \dots, k\}$, correspond to paths with the same top vertex. Lemma 4 implies that these paths also share the same maximum prefix of some length $x \in [0, M]$. Therefore, by Lemma 1, every edge incident to v_i corresponds to some SoC function whose first breakpoint has the x-coordinate x. Thus,

the leftmost point of the characteristic segment of each of these SoC functions lies on a vertical line defined by x; see Figure 4b. Similarly, edges incident to a bottom vertex $v_i \in V_1^-$ represent paths with the same maximum suffix of length $y \in [0, M]$. The last breakpoint of each SoC function associated with these paths lies on a horizontal line defined by the y-coordinate y. Hence, each of the k vertices defines either a vertical or a horizontal line. Every edge in the cycle C corresponds to a characteristic segment that starts at a vertical line and ends at a horizontal line, as shown in Figure 4b.

For a constant y inducing a horizontal line, we consider the leftmost x-coordinate of any breakpoint in an SoC function (corresponding to an edge in the cycle C) with the y-coordinate y; see Figure 4b. In total, we defined one x-coordinate for each vertex in C, denoted by x_i for $i \in \{1, \ldots, k\}$. Without loss of generality, assume $x_1 < x_2 < \cdots < x_k$. Then, we obtain $k-1$ intervals $[x_i, x_{i+1}]$, $i \in \{1, \ldots, k-1\}$. By assumption, every edge of C corresponds to a contributing path. Moreover, the characteristic segment of the SoC function of each contributing path is (partially) contained in the upper envelope of the SoC functions of all these paths (otherwise it would not contribute to the s–t profile). Given that all characteristic segments are parallel (with slope 1), this implies that each segment is the unique maximum over all characteristic segments on some interval $[x, x_{i+1}]$, $i \in \{1, \ldots, k-1\}$. However, there are only $k-1$ such intervals for k contributing paths; a contradiction.

Case 2. All edges represent bottom-top paths, and therefore $\{v_1, \ldots, v_k\} \subseteq V_1^- \cup V_2^+$. In this case, edges incident to a bottom vertex $v_i \in V_1^-$ for an $i \in \{1, \ldots, k\}$ correspond to paths with the same bottom vertex. By Lemma 4, these paths share the same minimum prefix with length $y \in [0, M]$. Moreover, observe that a contributing bottom-top path contains no subpath with cost below $-M$, since the cost of its maximum prefix must not exceed M. It follows that SoC functions of contributing bottom-top paths are of the form as in Case 3 of Lemma 1. Thus, the leftmost point of the characteristic segment of each SoC function represented by an edge incident to a bottom vertex $v_i \in V_1^-$ lies on the horizontal line defined by y. Similarly, edges incident to top vertices $v_i \in V_1^+$ correspond to characteristic segments whose rightmost point lies on the same vertical line defined by a constant $x \in [0, M]$. Along the lines of the first case, this yields a contradiction. ◀

4 Energy-Optimal Routes with Charging Stops

As battery capacities of EVs are typically rather small, *recharging* en route can be inevitable on long-distance trips. Therefore, we extend our model to incorporate charging stops. A subset $S \subseteq V$ of the vertices represents designated charging stations. Every station $v \in S$ has a predefined *SoC range* $R_v = [b_v^{\min}, b_v^{\max}] \subseteq [0, M]$. When arriving at v with *arrival SoC* b, we pick a desired *departure SoC* $b' \in [b_v^{\min}, b_v^{\max}] \cup \{b\}$ with $b \leq b'$. SoC ranges are useful to model user preferences or technical features of charging stations. For example, we set $R_v := [M, M]$ for battery swapping stations. It is always allowed to pick the arrival SoC b as departure SoC, to account for the possibility of not charging at v.

Given a source $s \in V$, a target $t \in V$, and the initial SoC $b_s \in [0, M]$ at s, the *Energy-Optimal Route with Charging Stops (EORCS)* problem asks for a feasible path that minimizes *overall* energy consumption, defined as the difference $b_s - b_t$ between SoC at source and target, plus the total amount r_t of energy recharged at charging stations $v \in S$ to reach t. Hence, our objective is to *maximize* $b_t - r_t$ among all feasible solutions.

There are certain challenging aspects to this problem. First, we have to determine the departure SoC when reaching a charging station with some arrival SoC. Note that it may

be wasteful to fully recharge the battery, as this may prevent recuperation on subsequent road segments. Second, subpaths of optimal paths are not optimal in general (e. g., detours to a charging station may be necessary). Hence, "greedy" choices can lead to suboptimal results. A natural way to overcome this issue is the use of label *sets* to keep multiple solutions at vertices, as in (exponential-time) multi-criteria search [23, 34]. In fact, this algorithm can be adapted to our problem setting, using labels with *continuous* ranges of SoC and recharged energy to reflect different choices at charging stations and the resulting SoC at a vertex. Then, vertices maintain labels that store an *SoC range* and a *charging range*. As in the multi-criteria scenario, we can apply Pareto dominance to remove suboptimal labels. However, it is not obvious whether such an approach has subexponential running time. Instead, we build upon tools from Section 3 to derive an alternative algorithm that maintains *single* labels on an extended search graph and that is conceptually simpler, can easily be integrated with known speedup techniques, and runs in polynomial time.

Optimal Paths between Charging Stations. When charging at a station $u \in S$, we have to ensure that the SoC is sufficient to reach t or the next charging station $v \in S$. Therefore, we examine an important subproblem, where given a charging station $u \in S$, an (optimal) arrival SoC b_u^{arr}, the amount r_u of energy recharged so far (at *previous* charging stations), and a vertex $v \in S \cup \{t\}$, we want to find a departure SoC $b_u^{\text{dep}} > b_u^{\text{arr}}$ that maximizes the objective at the target vertex t under the assumption that v is the next vertex where energy is recharged (or $v = t$ is the target itself). If we compute the u–v profile $f_{u,v}$, we can greedily optimize the objective on the s–v path by picking an SoC $b_u^{\text{dep}} > b_u^{\text{arr}}$ that maximizes $f(b_u^{\text{dep}}) - (r_u + r)$, where $r := b_u^{\text{dep}} - b_u^{\text{arr}}$ is the amount of energy charged at u. Unfortunately, the s–v path that maximizes this objective does not extend to the best solution at t in general. The reason for this is that charging too much energy might prevent the vehicle from recuperating energy on the following v–t path; see Figure 5 for an example.

Instead, we need a more sophisticated approach. To this end, we identify SoC values b_u^{dep} that may possibly lead to an optimal solution. We know (by the FIFO property [18]) that for an arbitrary departure SoC $b_u^{\text{dep}} \in [0, M]$, the optimal u–v subpath is also *energy-optimal* for b_u^{dep}. By Theorem 6, there can be at most $\mathcal{O}(|V|)$ such u–v paths. For each u–v path P contributing to the u–v profile $f_{u,v}$, we identify a (unique) *canonical* departure SoC b_P^{dep} at u that *always* optimizes the objective at t under the assumption that recharging is necessary at v (or $v = t$). Consider the SoC function f_P of P and let $b_P^{\min} := c(P_u^+)$ denote the minimum SoC that is necessary to traverse P, i. e., $f_P(b) = -\infty$ if and only if $b < b_P^{\min}$. Consequently, we have $b_P^{\text{dep}} \geq b_P^{\min}$. We also know that the objective $f_P(b_P^{\text{dep}}) - b_P^{\text{dep}} + b_u^{\text{arr}} - r_u$ of the s–v path can only *decrease* for $b_P^{\text{dep}} > b_P^{\min}$, since $b_u^{\text{arr}} - r_u$ is constant and the slope of f_P is at most 1 in the interval $[b_P^{\min}, M]$. Assuming that we recharge energy at v anyway, charging more than $b_P^{\min}$ will also never turn out to be essential after visiting v: If necessary, we can recharge missing energy at v. Therefore, given the SoC range $[b_u^{\min}, b_u^{\max}]$ of u, we pick the canonical departure SoC $b_P^{\text{dep}} := \max\{b_P^{\min}, b_u^{\min}\}$ for P, if this value lies in the SoC range of u. Otherwise, we have $b_u^{\max} < b_P^{\min}$ and charging at u never renders P feasible.

Search Graph Construction. Given the original graph G and the target vertex $t \in V$, we augment G with a *charging station (sub-)graph* $G_c = (V_c, E_c)$, which keeps separate vertices for distinct values of departure SoC at charging stations; see Figure 5. For each vertex $u \in S$, we create one *charging vertex* u' per distinct canonical departure SoC b_P^{dep} of *any* contributing path P from u to another charging station or to the target. We explicitly store the corresponding departure SoC b_P^{dep} with the vertex u', i. e., we keep a mapping

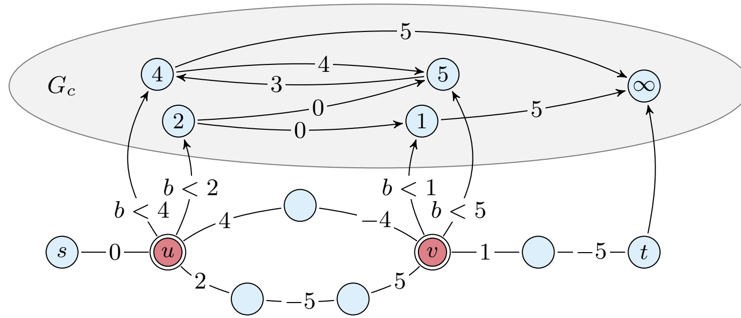


Figure 5 Search graph with charging stations (red, charging range $[0, M]$, $M = 5$). Vertex labels in G_c (shaded) indicate departure SoC. Edge labels indicate costs in G, arrival SoC in G_c, and SoC restrictions for transfer edges. While t is always reached from s with $b_t = 5$, the optimal path (and the objective $b_t - r_t$) depends on b_s: For $b_s < 2$, energy is charged at u ($b_u^{\text{dep}} = 2$) and v ($b_v^{\text{dep}} = 1$), which yields $r_t = 3 - b_s > 1$; for $b_s \in [2, 3]$ it is optimal to charge only at v ($b_v^{\text{dep}} = 1$) to get $r_t = 1$; for $b_s \in [3, 4)$ energy is only charged at v ($b_u^{\text{dep}} = 4$) to get $r_t = 4 - b_s \leq 1$; no charging is necessary at all for $b_s \geq 4$. In all cases the objective at v is maximized for $b_u^{\text{dep}} = 4$, which yields $b_v - r_v = b_u$.

$b^{\text{dep}}: V_c \to [0, M]$ and set $b^{\text{dep}}(u') := b_P^{\text{dep}}$. The vertex u' itself is added to V_c. We also add a dummy target vertex t' to V_c with $b^{\text{dep}}(t') := \infty$. For every contributing u–v path between vertices $u \in S$ and $v \in S \cup \{t\}$, we add edges (u', v') from the (unique) charging vertex $u' \in V_c$ of u with $b^{\text{dep}}(u') = b_P^{\text{dep}}$ to every corresponding vertex $v' \in V_c$ of v with $f_P(b_P^{\text{dep}}) < b^{\text{dep}}(v')$ to E_c. Together with the edge (u', v'), we also store the SoC upon arrival at v', i. e., we use a mapping $b^{\text{arr}}: E_c \to [0, M]$ and set $b^{\text{arr}}(u', v') := f_P(b_P^{\text{dep}})$. To connect G and G_c, we add (directed) *transfer edges* (v, v') from each charging station $v \in S \cup \{t\}$ to all its corresponding departure vertices $v' \in V_c$. Transfer edges have no cost, but may only be traversed if the current SoC is below the departure SoC $b^{\text{dep}}(v')$ of the respective departure vertex v'. Given the set E' of transfer edges, we obtain the *augmented graph* $G' := (V \cup V_c, E \cup E' \cup E_c)$. Note that its size is polynomial in the size of G.

A Polynomial-Time Algorithm. Using the augmented graph, we adapt EVD to find energy-optimal routes in G'. The modified algorithm maintains a single label $\ell(v)$ per vertex $v \in V \cup V_c$, which stores the values of SoC b_v and recharged energy r_v that maximize the objective $b_v - r_v$ at v. Initially, it sets $b_v = -\infty$ and $r_v = 0$ for all $v \in V$, except for the label $\ell(s) = (b_s, 0)$ of s. In each iteration of the main loop, the label $\ell(u) = (b_u, r_u)$ of some vertex u in G' with maximum key $b_u - r_u$ is extracted from the queue. If u is an original vertex, i. e., $u \in V$, the algorithm proceeds exactly like plain EVD by scanning its outgoing edges. If, additionally, u is a charging station, i. e., $u \in S$, its corresponding charging vertices $u' \in V_c$ are updated (and inserted into the queue) if $b_u < b^{\text{dep}}(u')$ and $\ell(u)$ yields an improvement to the label $\ell(u')$. Vertices $u \in V_c$ in the charging station graph are handled separately by the algorithm. For each outgoing edge (u, v) in G_c, a new label (b, r) is generated as follows. Its SoC is set to the arrival SoC $b := b^{\text{arr}}(u, v)$ at v and the amount of charged energy is set to $r := r_u + b^{\text{dep}}(u) - b_u$. If the resulting label (b, r) improves $\ell(v)$, the latter is updated accordingly. After termination, the label at the dummy target vertex t' (i. e., the unique vertex $t' \in V_c$ with $b^{\text{dep}}(t') = \infty$) contains the optimal pair of SoC and recharged energy. Making use of potential shifting, Theorem 7 follows directly from the polynomial size of G'.

▶ **Theorem 7.** *The problem EORCS can be solved in polynomial time.*

A Practical Variant. The construction of G_c is rather time-consuming on realistic instances. Luckily, we can move most work to preprocessing, since paths between charging stations are independent of source and target. We also propose a much simpler search graph, which can naturally be combined with CH for further speedup. We replace the graph G_c with an overlay graph $G_S = (S \cup \{t\}, S \times S \cup \{(v,t) \mid v \in S)\})$. Every edge (u,v) in G_S stores as its cost function the u–v profile (wrt. the original graph). Compared to G_c, this significantly reduces the number of vertices. Moreover, it is straightforward to construct G_S using profile search. We slightly modify the search algorithm to work with G_S instead of G_c: If a scanned vertex $u \in S$ represents a charging station, all shortcuts (u,v) in G_S are scanned. For each, the arrival SoC b that maximizes the objective at v is picked (such that $b \geq b_u$ and $b \in [b_u^{\min}, b_u^{\max}]$). If this yields an improvement to the label of v, it is updated accordingly.

As argued before, picking the SoC at v in this greedy fashion may lead to suboptimal results (recall Figure 5). On real-world networks, however, this is very unlikely to occur, as it requires an optimal route with two charging stops u and v, such that t can be reached from u via v, but *not* directly, and at the same charging too much energy at u (to reach v on an optimal s–v path) prevents recuperation along the v–t path due to a fully-charged battery. Consequently, our approach *always* produced optimal solutions in our tests; see Section 5.

Integration with CH. In its basic variant, CH [20] iteratively *contract* vertices in increasing order of (heuristic) importance during preprocessing, maintaining distances between all remaining vertices by adding *shortcut* edges, if necessary. *Witness searches* determine whether a shortcut is required to preserve distances. The CH query runs bidirectional from source and target on the input graph augmented by all shortcuts added during preprocessing, following only *upward* edges (from less important to more important vertices).

When solving EORCS with CH, we do not contract charging stations during preprocessing [7, 41]. Hence, we stop vertex contraction at some point, leaving an uncontracted *core* of charging stations (and possibly other vertices). We run profile searches on this (relatively small) core graph to quickly construct the overlay G_S. Shortcuts are only added to G_S if their corresponding SoC function is *finite* for some SoC.

In a basic approach, witness search uses profile search to determine whether a shortcut is necessary. For faster preprocessing, an alternative approach replaces profiles by scalar *upper bounds* $\max_{b \in [0,M]}(b - f(b))$ on the energy consumption of an edge with SoC function f. Observe that negative costs are ruled out this way, since consumption must be at least zero for a fully charged battery. This re-enables Dijkstra's algorithm for witness searches, computing upper bounds $a \in [0, M]$ on energy consumption between a given pair of vertices. A shortcut candidate is inserted only if its SoC function f consumes less energy for at least one SoC, i. e., $b - f(b) < a$ for some $b \in [0, M]$. When using upper bounds, we may end up inserting unnecessary shortcuts. This does not affect correctness, but may (slightly) slow down queries. (Similarly, Eisner et al. [18] use a sampling approach to avoid costly profile search during preprocessing in their implementation.)

To obtain the full path description, we enable path unpacking by storing via vertices during contraction, as in plain CH [20]. (Note, however, that we need one via vertex per contributing path of an SoC function.) Additionally, we have to reconstruct paths representing shortcuts between charging stations within the core. This can be done by precomputing and storing the paths explicitly (in the core), or by running a profile search between each consecutive pair of charging stations in the optimal path. Finally, the retrieved paths in the core are unpacked. The optimal amount of energy that must be recharged is easily obtained from the SoC profiles in the overlay G_S (by picking a departure SoC for each profile that maximizes the objective).

The query algorithm consists of two phases. The first runs a *backward* profile search from t, scanning only upward and edges in the core. Shortcuts in G_S are ignored by this search. After its termination, SoC profiles from each charging station to the target are known. We (temporarily) add the target and all corresponding shortcuts to G_S. The second phase runs modified EVD from the source s with initial SoC b_s, on a graph consisting of upward edges and all edges in the core (including G_S).

Adding A* Search. On large instances, scanning shortcuts in the dense subgraph G_S becomes the major bottleneck of the approach. We add A* search [24] to improve performance. The basic idea of A* search is to compute a *consistent potential function* $\pi \colon V \to \mathbb{R}$ on the vertices, which fulfills the condition $c(u,v) - \pi(u) + \pi(v) \geq 0$ for all edges $(v,v) \in E$. The potential of a vertex is added to the key of a label when updating the priority queue. To make the search goal directed, we compute a consistent potential function where vertices that are closer to the target have smaller keys.

Before the CH search, we run a label-correcting backward search from t, scanning upward edges and core edges (except for shortcuts in G_S), with (scalar) *lower bounds* on energy consumption as edge costs. This yields, for each vertex v in the core, a lower bound $\underline{c}(v)$ on energy consumption from v to t. Moreover, $\underline{c}(\cdot)$ induces a consistent potential function on V, as follows immediately from the triangle inequality. The forward search is then split into two phases. The first scans upward edges, but ignores outgoing edges of core vertices. The second phase is initialized with all core vertices scanned during the first phase. Using the potential function, this phase becomes *goal directed*.

An aggressive variant of A* search achieves further speedup at the cost of suboptimal results. As before, when a charging station is visited by the forward search, all outgoing shortcuts (u,v) in G_S are scanned. However, we update the label of at most *one* vertex $v \in S$ and insert it into priority queue, namely, the one with maximum key among all vertices that are improved by the scans.

5 Experiments

We implemented all approaches in C++, using g++ 4.8.5 (-O3) as compiler. Experiments were conducted on a single core of a 4-core Intel Xeon E5-1630v3 clocked at 3.7 GHz, with 128 GiB of DDR4-2133 RAM, 10 MiB of L3, and 256 KiB of L2 cache.

Input and Methodology. We ran experiments on a graph representing the road network of Western Europe, kindly provided by PTV AG (`http://ptvgroup.com`). Energy consumption data stems from PHEM, a detailed micro-scale emission model [25]. We consider two vehicle models. The first is based on a real production vehicle (Peugeot iOn) with a battery capacity of 16 kWh (corresponding to a range of roughly 100–150 km). The second is an artificial model with a capacity of 85 kWh (400–500 km range, similar to recent Tesla models). Data source mentioned above are proprietary, but enable us to test our algorithms on detailed and realistic input data. To accurately assign consumption values to road segments, we retrieved road slopes based on elevation data from the freely available SRTM dataset (`http://www2.jpl.nasa.gov/srtm`). Removing edges without reasonable energy consumption (e. g., due to large areas with missing elevation data), we obtain a graph with 22 198 628 vertices and 51 088 095 edges after extracting the largest strongly connected component from the remaining graph [9]. About 11.8 % and 15.2 % of these edges have negative cost for the Peugeot and the artificial model, respectively. We also conduct

Table 1 Performance of our approaches (Europe). The columns G_S, CH, A*, and agg. (aggressive A*) indicate whether a technique is enabled (●) or not (○). For each approach and model, we report preprocessing time, number of vertex scans during queries (# V. Sc.), and query times.

Techniques				Peugeot, 16 kWh			Artificial, 85 kWh		
G_S	CH	A*	agg.	Prepr. [s]	# V. Sc.	Q. [ms]	Prepr. [s]	# V. sc.	Q. [ms]
○	○	○	○	–	8 895 038	20 160.9	–	11 033 760	32 928.8
●	○	○	○	1 487	759 951	710.0	15 062	7 753 601	6 285.7
●	●	○	○	2 860	8 433	309.6	3 246	19 616	1 281.5
●	●	●	○	2 860	3 563	128.2	3 246	10 418	297.5
●	●	●	●	2 860	1 599	41.0	3 246	9 579	157.8

Table 2 Performance for different distributions and types of charging stations (Germany, Peugeot). Besides timings for preprocessing and queries, we report the number of charging stations ($|S|$), edges in G_S ($|E_S|$), and vertex scans (# V. Sc.) and edge scans (# E. Sc.) during queries.

		Prepr.		Queries		
Scenario	$\|S\|$	T. [s]	$\|E_S\|$	# V. Sc.	# E. Sc.	T. [ms]
reg-cm	1 966	548.5	539 145	4 592	125 535	4.22
mix-cm	1 966	548.1	539 145	4 592	125 381	4.19
reg-r0.01	469	487.2	22 231	2 234	50 070	1.30
reg-r0.1	4 692	582.7	2 263 310	8 904	223 779	7.97
reg-1.0	46 920	965.0	227 514 459	60 527	1 828 581	73.46

experiments on the subnetwork of Germany, which has 4 692 091 vertices and 10 805 429 edges. We located 13 810 charging stations (1 966 of them in Germany) on ChargeMap (`chargemap.com`). Unless mentioned otherwise, all reported query times are average values of 1 000 queries, with source and target vertices picked uniformly at random. Charging stations have the SoC range $[0, M]$ and the initial SoC is set to $b_s = M$.

Evaluation. Table 1 compares different approaches to solve EORCS on our main test instance (Europe) for both vehicle models. Applied techniques are indicated by the four leftmost columns. The first line (no speedup technique enabled) shows our exact baseline approach, which is based on bi-criteria search. It requires no preprocessing, but takes 20–30 seconds to answer queries, which is rather impractical. Simply using the charging station graph already reduces query times greatly. However, scalability of this approach is limited, as increasing the vehicle range affects both preprocessing (longer paths between charging stations must be computed) and queries (the search in the uncontracted network dominates running times). Integrating CH clearly pays off, as it significantly reduces the number of vertex scans and query time after moderate preprocessing effort (below an hour). Query time is dominated by the search in G_S. A* search helps to reduce effort spent searching in G_S and makes our approach rather practical, with running times of less than 300 ms for the artificial model. Moreover, note that even though we use a (formally) inexact implementation, the optimal solution is found in *all* queries. The aggressive variant of A* further reduces query times at the cost of inexact results (even in practice). The average relative error (not reported in the table) is 0.7 % for the Peugeot model and less than 0.01 % for the artificial one. This discrepancy in relative error can be explained by the fact that a larger battery allows the EV

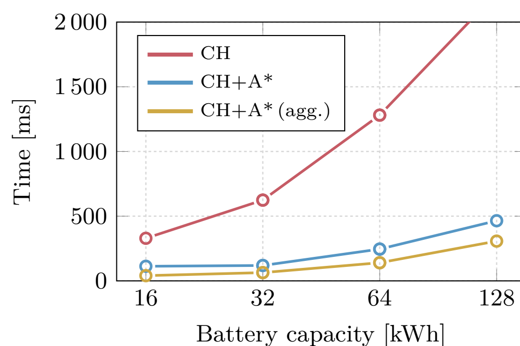


Figure 6 Algorithm performance subject to cruising range (Europe, Peugeot). Each point is the median running time of 1 000 queries for one of the different approaches (CH, CH with A*, CH with aggressive A*) under varying battery capacities.

to stick to energy-optimal paths (fewer detours are necessary), so the quality of the bounds used in A* search increases. Consequently, outliers for the Peugeot scenario exceed 10 % in relative error in about 1 % of the cases, while even the maximum error is below 0.5 % for the artificial model. For all techniques, the artificial model is harder to solve. This is mostly due to the dense charging station graph (more labels per vertex for the baseline approach), since more charging stations are reachable from each station.

In Table 2, we evaluate the performance of our fastest empirically exact approach (CH with A* search) under varying types and distributions of charging stations. The first scenario (reg-cm) uses stations from ChargeMap with (default) SoC range $[0, M]$. The second (mix-cm) uses the same stations, but assigns each a charging range of a regular station ($[0, M]$), a "super charger" that quickly charges to 80 % SoC ($[0, 0.8M]$), or a swapping station ($[M, M]$), with equal probability. The results indicate that SoC ranges have little impact on performance. This is not surprising, since restricting the departure SoC can only reduce the search space (the effect is negligible, though). Finally, we consider random distributions of charging stations with default SoC ranges (reg-r0.01, reg-r0.1, reg-1.0), where we pick 0.01 %, 0.1 %, and 1.0 % of the vertices in V as charging stations uniformly at random, respectively. The number of charging stations has a significant impact on algorithm performance. Given that the number of edges in G_S grows quadratically in the number of charging stations, preprocessing and query slow down for very dense networks of charging stations. This limits scalability, but our approach easily handles realistic distributions of charging stations (for the scenario reg-1.0, the number of charging stations is higher than the current number of gas stations in Germany).

Finally, Figure 6 shows running times of our algorithms for different battery capacities. Without A* search, running time roughly doubles with battery capacity, because G_S becomes more dense (more reachable charging stations). Adding A* search, scalability with available cruising range improves, since potentials quickly guide the search towards the target.

6 Conclusion

We examined consumption profiles for EVs and proved that their complexity is at most linear in the graph size. We also investigated energy-optimal routes with charging stops and showed how profile search can be utilized to solve the problem in polynomial time. In a sense, we closed the gap between (efficiently solvable) energy-optimal routes [18, 36]

and $\mathcal{NP}$-hard time-constrained variants with charging stops [7, 35] (which generalize the problem setting considered in this work). In particular, it is indeed the addition of a second optimization criterion (travel time) that makes the latter problems $\mathcal{NP}$-hard, rather than the incorporation of charging stations in combination with battery constraints. We also proposed a practical variant, which computes optimal results in well below a second on realistic, large-scale networks.

Interesting lines of future work include reducing the number of edges in the overlay of charging stations for better performance and scalability [13, 26, 39] or integration of Customizable CH [16] for faster preprocessing. Moreover, one could consider a *profile* variant of EORCS, i. e., ask for a consumption profile instead of a single path.

References

1. Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The Optimal Routing Problem in the Context of Battery-Powered Electric Vehicles. In *Proceedings of the 2nd CPAIOR Workshop on Constraint Reasoning and Optimization for Computational Sustainability (CROCS'10)*, 2010.
2. Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The Shortest Path Problem Revisited: Optimal Routing for Electric Vehicles. In *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence (KI'10)*, volume 6359 of *Lecture Notes in Computer Science*, pages 309–316. Springer, 2010.
3. Mikhail J. Atallah. Some Dynamic Computational Geometry Problems. *Computers & Mathematics with Applications*, 11(12):1171–1181, 1985.
4. Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Route Planning in Transportation Networks*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.
5. Gernot V. Batz and Peter Sanders. Time-Dependent Route Planning with Generalized Objective Functions. In *Proc. of the 20th Annual European Symp. on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2012.
6. Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15:2.3:1–2.3:31, 2010.
7. Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles. In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'15)*, pages 44:1–44:10. ACM, 2015.
8. Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner. Speed-Consumption Tradeoff for Electric Vehicle Route Planning. In *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*, volume 42 of *OpenAccess Series in Informatics (OASIcs)*, pages 138–151. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014.
9. Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-Optimal Routes for Electric Vehicles. In *Proc. of the 21st ACM SIGSPATIAL Int'l Conference on Advances in Geographic Information Systems (GIS'13)*, pages 54–63. ACM, 2013.
10. Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Dynamic Time-Dependent Route Planning in Road Networks with User Preferences. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*, volume 9685 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2016.
11. Brian C. Dean. Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms. Technical report, Massachusetts Institute of Technology, 2004.

12 Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, 2011.

13 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks. *Transportation Science*, 2015.

14 Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2007.

15 Daniel Delling and Dorothea Wagner. *Time-Dependent Route Planning*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

16 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 21:1.5:1–1.5:49, 2016.

17 Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

18 Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal Route Planning for Electric Vehicles in Large Networks. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1108–1113. AAAI Press, 2011.

19 Luca Foschini, John Hershberger, and Subhash Suri. On the Complexity of Time-Dependent Shortest Paths. *Algorithmica*, 68(4):1075–1097, 2014.

20 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

21 Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

22 Michael T. Goodrich and Paweł Pszona. Two-Phase Bicriterion Search for Finding Fast and Efficient Electric Vehicle Routes. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'14)*, pages 193–202. ACM, 2014.

23 Pierre Hansen. Bicriterion Path Problems. In *Multiple Criteria Decision Making – Theory and Application*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127. Springer, 1980.

24 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

25 Stefan Hausberger, Martin Rexeis, Michael Zallinger, and Raphael Luz. Emission Factors from the Model PHEM for the HBEFA Version 3. Technical report I-20/2009, University of Technology, Graz, 2009.

26 Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13:2.5:1–2.5:26, 2009.

27 Gerhard Huber and Klaus Bogenberger. Long-Trip Optimization of Charging Strategies for Battery Electric Vehicles. *Transportation Research Record: Journal of the Transportation Research Board*, 2497:45–53, 2015.

28 Donald B. Johnson. A Note on Dijkstra's Shortest Path Algorithm. *Journal of the ACM*, 20(3):385–388, 1973.

29 Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13, 1977.

30 Sebastian Kluge, Claudia Sánta, Stefan Dangl, Stefan M. Wild, Martin Brokate, Konrad Reif, and Fritz Busch. On the Computation of the Energy-Optimal Route Dependent on the Traffic Load in Ingolstadt. *Transportation Research Part C: Emerging Technologies*, 36:97–115, 2013.

31 Yuichi Kobayashi, Noboru Kiyama, Hirokazu Aoshima, and Masamori Kashiyama. A Route Search Method for Electric Vehicles in Consideration of Range and Locations of Charging Stations. In *Proceedings of the 7th IEEE Intelligent Vehicles Symposium (IV'11)*, pages 920–925. IEEE, 2011.

32 Chung-Shou Liao, Shang-Hung Lu, and Zuo-Jun Max Shen. The Electric Vehicle Touring Problem. *Transportation Research Part B: Methodological*, 86:163–180, 2016.

33 Chensheng Liu, Jing Wu, and Chengnian Long. Joint Charging and Routing Optimization for Electric Vehicle Navigation Systems. In *Proceedings of the 19th International Federation of Automatic Control World Congress (IFAC'14)*, volume 47 of *IFAC Proceedings Volumes*, pages 9611–9616. Elsevier, 2014.

34 Ernesto Q. V. Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

35 Sören Merting, Christian Schwan, and Martin Strehler. Routing of Electric Vehicles: Constrained Shortest Path Problems with Resource Recovering Nodes. In *Proceedings of the 15th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'15)*, volume 48 of *OpenAccess Series in Informatics (OASIcs)*, pages 29–41. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015.

36 Martin Sachenbacher, Martin Leucker, Andreas Artmeier, and Julian Haselmayr. Efficient Energy-Optimal Routing for Electric Vehicles. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1402–1407. AAAI Press, 2011.

37 René Schönfelder and Martin Leucker. Abstract Routing Models and Abstractions in the Context of Vehicle Routing. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 2639–2645. AAAI Press, 2015.

38 René Schönfelder, Martin Leucker, and Sebastian Walther. Efficient Profile Routing for Electric Vehicles. In *Proceedings of the 1st International Conference on Internet of Vehicles (IOV'14)*, volume 8662 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2014.

39 Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering & Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.

40 Olivia J. Smith, Natashia Boland, and Hamish Waterer. Solving Shortest Path Problems with a Weight Constraint and Replenishment Arcs. *Computers & Operations Research*, 39(5):964–984, 2012.

41 Sabine Storandt. Quick and Energy-Efficient Routes: Computing Constrained Shortest Paths for Electric Vehicles. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'12)*, pages 20–25. ACM, 2012.

42 Sabine Storandt and Stefan Funke. Cruising with a Battery-Powered Vehicle and Not Getting Stranded. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12)*, pages 1628–1634. AAAI Press, 2012.

43 Zhonghao Sun and Xingshe Zhou. To Save Money or to Save Time: Intelligent Routing Design for Plug-In Hybrid Electric Vehicle. *Transportation Research Part D: Transport and Environment*, 43:238–250, 2016.

44 Timothy M. Sweda, Irina S. Dolinskaya, and Diego Klabjan. Adaptive Routing and Recharging Policies for Electric Vehicles. Working paper no. 14-02, Northwestern University, Illinois, 2014.

45 Yan Wang, Jianmin Jiang, and Tingting Mu. Context-Aware and Energy-Driven Route Optimization for Fully Electric Vehicles via Crowdsourcing. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1331–1345, 2013.

46 Ady Wiernik and Micha Sharir. Planar Realizations of Nonlinear Davenport-Schinzel Sequences by Segments. *Discrete & Computational Geometry*, 3(1):15–47, 1988.

Efficient Traffic Assignment for Public Transit Networks*

Lars Briem[1], Sebastian Buck[2], Holger Ebhart[3], Nicolai Mallig[4], Ben Strasser[5], Peter Vortisch[6], Dorothea Wagner[7], and Tobias Zündorf[8]

1 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
lars.briem@kit.edu,

2 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
sebastian.buck@kit.edu

3 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
holger.ebhart@ira.uka.de

4 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
nicolai.mallig@kit.edu

5 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
strasser@kit.edu

6 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
peter.vortisch@kit.edu

7 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
dorothea.wagner@kit.edu

8 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
zuendorf@kit.edu

Abstract

We study the problem of computing traffic assignments for public transit networks: Given a public transit network and a demand (i.e. a list of passengers, each with associated origin, destination, and departure time), the objective is to compute the utilization of every vehicle. Efficient assignment algorithms are a core component of many urban traffic planning tools. In this work, we present a novel algorithm for computing public transit assignments. Our approach is based upon a microscopic Monte Carlo simulation of individual passengers. In order to model realistic passenger behavior, we base all routing decisions on travel time, number of transfers, time spent walking or waiting, and delay robustness. We show how several passengers can be processed during a single scan of the network, based on the Connection Scan Algorithm [6], resulting in a highly efficient algorithm. We conclude with an experimental study, showing that our assignments are comparable in terms of quality to the state-of-the-art. Using the parallelized version of our algorithm, we are able to compute a traffic assignment for more than ten million passengers in well below a minute, which outperforms previous works by more than an order of magnitude.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases Algorithms, Optimization, Route planning, Public transportation

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.20

* Tobias Zündorf's research was supported by DFG Research Grant WA 654/23-1.

© Lars Briem, Sebastian Buck, Holger Ebhart, Nicolai Mallig, Ben Strasser, Peter Vortisch, Dorothea Wagner, and Tobias Zündorf;
licensed under Creative Commons License CC-BY

16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 20; pp. 20:1–20:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Traffic assignment problems are widely studied for the case of street networks and private transport. However, virtually no work addresses the algorithmic challenges of finding a traffic assignment for public transit networks, despite the fact that such assignments are a very important tool for planning public transportation services. When implementing new public transit lines (or redirecting existing ones), it is often desired, that the capacity of all the vehicles serving the lines is well utilized. If vehicles are overcrowded, then more or larger vehicles have to be deployed. However, if vehicles are only sparsely used, they may be dropped from the schedule. Using traffic assignments, the utilization of the vehicles can be estimated ahead of time allowing an efficient public transportation service design.

Determining a public transit assignment requires two components: The timetable of the underlying public transit network; and an estimation of the overall passenger flow, specified by individual origin, destination, and time triples, called demand. The most basic variant of the assignment problem asks only for the expectable utilization of the vehicles operating in the public transit network. A more elaborate variant of the problem requires that every individual passenger is assigned to a route from his origin to his destination. The objective of the assignment is in every case a statistical analysis of the network utilization. Thus, a passenger may even be proportionally assigned to several routes, in order to increase the overall accuracy of the assignment.

In this work we present a novel algorithm for computing public transit assignments. Our approach for assigning routes to individual passengers is based on a Monte Carlo simulation. Every passenger's movement through the network is simulated step by step until his destination is reached. We consider multiple criteria of each possible route, in order to achieve a realistic movement of the passengers: the arrival time at the destination, the time that has to be waited for connecting vehicles, the time that has to spent walking between stops, the number of transfers between vehicles, and the delay robustness. We describe an efficient approach for this simulation, based on the Connection Scanning Algorithm [6], which allows for efficient one to all queries on public transit networks.

Our paper is organized as follows: In Section 2 we formally define the public transit network and the demand, and we introduce the basic notation used throughout this paper. Next, we propose the notion of perceived arrival times, which we use to model passenger preferences in Section 3. We continue with presenting our algorithm in Section 4. In Section 5 we conduct an experimental study, showing that the quality of our assignment is comparable to state-of-the-art, while the running time is more than an order of magnitude lower.

1.1 Related Work

The problem of finding a traffic assignment comprises two important subproblems. First, achieving a high quality assignment requires a sophisticated procedure for deciding which route a passenger would choose in the real world. Second, efficient route planning algorithms are required in order to compute possible routes to choose from. There has been a lot of research addressing route planning problems in recent years. A comprehensive overview of state-of-the art route planning algorithms is given in [2]. Unlike time independent route planning, it is quite hard to accelerate routing algorithms for public transit [4]. Several speed-up techniques have been proposed, in order to increase the efficiency of public transit route planning, many of them exploit the special structure of timetables. The RAPTOR [5] algorithm is one of the first techniques solely based on an efficient timetable representation. With Transfer Patterns [1, 3] a first approach that utilizes preprocessing in order to enable fast

public transit queries was introduced. The author of [12] proposes an algorithm for fast profile queries, based on a data model focused on trips and transfers between them. The Connection Scanning Algorithm (CSA) [6, 11] relies on a particularly simple data model, namely a sorted array of connection. Nevertheless, it allows for fast one to all profile queries. Based on CSA, the MEAT [7] technique was developed, enabling delay robust journey planning.

An overview over traffic assignment techniques and models can be found in [10]. An important concept for achieving realistic traffic assignments are equilibrium models, which enable that the assignment adapts to congested parts of the network. Various variants of equilibrium models are discussed in [8]. Just like route planning, traffic assignment problems become more difficult when applied to public transit networks. An implementation of state-of-the-art traffic assignment algorithms is available in VISUM from PTV AG[1].

2 Preliminaries

Our algorithm operates on a public transit network $(\mathcal{C}, G)$ consisting of a finite set of elementary connections $\mathcal{C}$ and a directed, weighted *transfer graph* $G = (\mathcal{V}, \mathcal{E}, \tau_{\text{trans}})$.

A *connection* $c \in \mathcal{C}$ is a tuple $(v_{\text{dep}}(c), v_{\text{arr}}(c), \tau_{\text{dep}}(c), \tau_{\text{arr}}(c), \text{trip}(c))$ representing a vehicle driving from a *departure stop* $v_{\text{dep}}(c) \in \mathcal{V}$ to an *arrival stop* $v_{\text{arr}}(c) \in \mathcal{V}$ without any intermediate stops. The vehicle is scheduled to depart from $v_{\text{dep}}(c)$ at the *departure time* $\tau_{\text{dep}}(c)$ and arrives at $v_{\text{arr}}(c)$ at the *arrival time* $\tau_{\text{arr}}(c)$ which we require to be greater than $\tau_{\text{dep}}(c)$. We assume that connections departing from the same stop have a well defined order of departure, i.e. $v_{\text{dep}}(c) = v_{\text{dep}}(c') \Rightarrow \tau_{\text{dep}}(c) \neq \tau_{\text{dep}}(c')$. However, this is not a real restriction, as such a scenario is unrealistic, and a unique order could be established by perturbing the departure times by some $\varepsilon > 0$. Consecutive connections c_1 and c_2 are part of a *trip* $\text{trip}(c_1) = \text{trip}(c_2)$ if they are served by the same vehicle. Using two successive connections of different trips requires a transfer in between.

Valid *transfers* are defined using the transfer graph $G = (\mathcal{V}, \mathcal{E}, \tau_{\text{trans}})$, where $\mathcal{V}$ is a set of *vertices*, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of *directed edges*, and $\tau_{\text{trans}} \colon \mathcal{E} \to \mathbb{N}_0$ is an edge weight representing the minimal required *transfer time* for using an edge. We expand τ_{trans} for arbitrary pairs of vertices $u, v \in \mathcal{V}$, by setting $\tau_{\text{trans}}(u, v) := \tau_{\text{trans}}(e)$ if there exists an edge $e = (u, v) \in \mathcal{E}$, otherwise we define $\tau_{\text{trans}}(u, v) := \infty$. Transferring between connections c_1 and c_2 of different trips is only possible if there exists an edge $e = (v_{\text{arr}}(c_1), v_{\text{dep}}(c_2))$ connecting the arrival stop of the first connection c_1 with the departure stop of the second connection c_2, such that $\tau_{\text{arr}}(c_1) + \tau_{\text{trans}}(e) \leq \tau_{\text{dep}}(c_2)$. Note that by our definition, a transfer edge has to be used even if $v_{\text{arr}}(c_1) = v_{\text{dep}}(c_2)$. This enables us to model minimum transfer times for changing between trips stopping at the same vertex. We define by $\tau_{\text{wait}}(v, \tau, c) := \tau_{\text{dep}}(c) - \tau - \tau_{\text{trans}}(v, v_{\text{dep}}(c))$ the additional waiting time after transferring from vertex v at time τ to $v_{\text{dep}}(c)$, before c departs. Given two connections $c, c' \in \mathcal{C}$, the waiting time for transferring from c to c' is given by $\tau_{\text{wait}}(c, c') := \tau_{\text{wait}}(v_{\text{arr}}(c), \tau_{\text{arr}}(c), c')$. Thus, a transfer between connections c and c' is valid if and only if $\tau_{\text{wait}}(c, c') \geq 0$ holds. Following [5, 11], we require that the transfer graph is transitively closed and fulfills the triangle inequality. This ensures that an edge $e = (u, v)$ is always a shortest path in G from u to v. We call a vertex $v \in \mathcal{V}$ *stop* if there exists at least one connection c such that $v_{\text{dep}}(c) = v$ or $v_{\text{arr}}(c) = v$, the set of all stops is denoted by $\mathcal{S} \subseteq \mathcal{V}$.

A *journey* $j = \langle c_1, \ldots, c_k \rangle$ is a sequence of connections, such that transferring between subsequent connections in j is valid. Formally, this means that the connections in j have to be

[1] `http://vision-traffic.ptvgroup.com/en-us/products/ptv-visum/`

sorted chronologically. Furthermore, we require that subsequent connections c_i and c_{i+1} are either part of the same trip (i.e. $\mathrm{trip}(c_i) = \mathrm{trip}(c_{i+1})$), or there exits a transfer connecting them (i.e. $\tau_{\mathrm{wait}}(c_i, c_{i+1}) \geq 0$), for every $i \in [1, k-1]$. Analogous to connections, we define a departure stop $v_{\mathrm{dep}}(j) := v_{\mathrm{dep}}(c_1)$, a departure time $\tau_{\mathrm{dep}}(j) := \tau_{\mathrm{dep}}(c_1)$, an arrival stop $v_{\mathrm{arr}}(j) := v_{\mathrm{arr}}(c_k)$, and an arrival time $\tau_{\mathrm{arr}}(j) := \tau_{\mathrm{arr}}(c_k)$, for a journey $j = \langle c_1, \dots, c_k \rangle$.

In addition to a public transit network, the input of a traffic assignment instance also contains a set of demands $\mathcal{D}$. A demand $D \in \mathcal{D}$ is a triple $(o(D), d(D), \tau_{\mathrm{dep}}(D))$ representing a passenger who wishes to travel from his *origin* $o(D) \in \mathcal{V}$ to his *destination* $d(D) \in \mathcal{V}$, starting at his departure time $\tau_{\mathrm{dep}}(D)$. The objective of the traffic assignment is to compute for every passenger represented by $\mathcal{D}$ a journey that satisfies his demand. A journey j *satisfies* a demand D, if it can be used to travel from $o(D)$ to $d(D)$, with a departure time of at least $\tau_{\mathrm{dep}}(D)$. More precisely, the journey must be reachable from the demanded origin when departing after $\tau_{\mathrm{dep}}(D)$. This is the case, if either the journey departs not earlier than $\tau_{\mathrm{dep}}(D)$ directly from $o(D)$ (i.e. $o(D) = o(j) \wedge \tau_{\mathrm{dep}}(D) \leq \tau_{\mathrm{dep}}(j)$) or if transferring from the origin to the journeys departure is valid (i.e. $\tau_{\mathrm{dep}}(D) + \tau_{\mathrm{trans}}(o(D), v_{\mathrm{dep}}(j)) \leq \tau_{\mathrm{dep}}(j)$). Furthermore, the journey has to end at the demanded destination, either directly or by using an additional transfer. Formally, we assume that either $v_{\mathrm{arr}}(j) = d(D)$ or $\tau_{\mathrm{trans}}(v_{\mathrm{arr}}(j), d(D)) < \infty$ holds. An empty journey $j = \langle\rangle$ satisfies a demand D, if $o(D) = d(D)$ or $\tau_{\mathrm{trans}}(o(D), d(D)) < \infty$ holds. Note that we do not require a journey to be optimal with respect to any metric, in order to satisfy a demand.

Given two vertices $u, v \in \mathcal{V}$, a *u-v-profile* is a function $f^{u,v}(\tau)$ mapping departure times τ onto the minimal costs for traveling from u at time τ to v, with respect to some cost function. If no such journey exists we define $f^{u,v}(\tau)$ as ∞. Profile functions are piecewise linear, since for every departure time τ the value $f^{u,v}(\tau)$ corresponds to a unique journey. Each journey contributing to $f^{u,v}(\tau)$ is either empty or has a fixed departure time, depending solely on the journeys first connection. Since there exists only a finite number of connections, every profile function can be described using a finite number of supporting points.

3 Perceived Arrival Time (PAT)

The core problem of computing a public transit assignment, is to decide for each passenger which connections he takes in order to reach his destination. The quality of the resulting assignment highly depends on these choices. Thus it is important to model the behavior and preferences of the passengers in a realistic way. This means that we cannot assign connections to the passengers solely based on the travel time of the resulting journey. For example, a passenger might prefer a journey with a slightly longer travel time, if this reduces the number of changes between vehicles.

As a means of reflecting the passengers preferences, we introduce the notion of *perceived arrival time* (short PAT). Given a connection $c \in \mathcal{C}$ and a destination $d \in \mathcal{V}$, the perceived arrival time $\tau^{\mathrm{p}}(c, d)$ is a measurement for how useful c is in order do reach d. The PAT $\tau^{\mathrm{p}}(c, d)$ depends on the possible journeys that end at d and contain c. We consider five properties of these journeys that influence the perceived arrival time: the actual arrival time at d, the number of transfers, the time spend walking, the time spend waiting, and the delay robustness. We account for walking and waiting time by weighting the corresponding times with factors $\lambda_{\mathrm{walk}}, \lambda_{\mathrm{wait}} \in \mathbb{R}$. For every transfer during the journey we add an additional cost of $\lambda_{\mathrm{trans}} \in \mathbb{R}$. Finally, we incorporate delay robustness by computing the expected arrival time under the assumption that each connection has a random delay of at most $\Delta_{\tau}^{\max}$.

We adapt the concept of minimum expected arrival time (MEAT) introduced by Dibbelt et al. [7], in order to model delay robustness. Following their approach, we introduce a

random variable $\Delta_\tau^c \in \mathbb{R}_0^+$ for every connection $c \in \mathcal{C}$, that represents the delay of the connection. This means that the arrival stop $v_{\text{arr}}(c)$ will be reached at $\tau_{\text{arr}}(c) + \Delta_\tau^c$. Thus, transferring to another connection can become invalid, if the delay exceeds the waiting time of the transfer. The probability that the delay is at most x, is given by the cumulative distribution function $P[\Delta_\tau^c \leq x]$. We define $P[\Delta_\tau^c \leq x]$ as follows: $P[\Delta_\tau^c \leq x] := 0$ for $x \leq 0$, $P[\Delta_\tau^c \leq x] := 1$ for $x \geq \Delta_\tau^{\max}$, and $P[\Delta_\tau^c \leq x] := 31/30 - (11\Delta_\tau^{\max})/(300x + 30\Delta_\tau^{\max})$ for $0 < x < 1$, where $\Delta_\tau^{\max}$ is the maximal delay that can occur. Based on this, the probability that a transfer between two connections $c, c' \in \mathcal{C}$ is valid, is given by $P[\Delta_\tau^c \leq \tau_{\text{wait}}(c, c')]$. Additionally, we define the probability $P[y < \Delta_\tau^c \leq x] := P[\Delta_\tau^c \leq x] - P[\Delta_\tau^c \leq y]$ that the delay of c is between y and x. For more details on the delay model see [7].

We now proceed with defining the perceived arrival time $\tau^{\text{p}}(c, d)$ in a recursive way, which allows us to take all journeys containing c into account. There exist three distinct cases for continuing a journey after using the connection c. If it is possible to use a transfer from the arrival stop of c to the destination, then the journey can be completed by walking. Otherwise, the journey continues either with the next connection of the same trip as $\text{trip}(c)$ or the vehicle serving c is left at $v_{\text{arr}}(c)$. Therefore we define

$$\tau_{\text{arr}}^{\text{p}}(c, d) := \min\{\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{walk}),\ \tau_{\text{arr}}^{\text{p}}(c, d \mid \text{trip}),\ \tau_{\text{arr}}^{\text{p}}(c, d \mid \text{trans})\},$$

where $\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{walk})$ is the PAT under the constraint that the journey is completed by walking from c to d, $\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{trip})$ is the PAT under the constraint that the journey continues with the same trip as $\text{trip}(c)$, and $\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{trans})$ is the PAT under the constraint for that the journey continues with a transfer to another connection after c. The perceived arrival time for walking to the destination is defined as:

$$\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{walk}) := \begin{cases} \tau_{\text{arr}}(c) & \text{if } v_{\text{arr}}(c) = d \\ \tau_{\text{arr}}(c) + \lambda_{\text{walk}} \cdot \tau_{\text{trans}}\big(v_{\text{arr}}(c), d\big) & \text{otherwise.} \end{cases}$$

This means that the PAT is the actual arrival time, if the destination is reached directly by using c. If this is not the case, the time needed for walking to the destination is multiplied with the cost factor λ_{walk} and added to the arrival time. For the definition of $\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{trip})$ let $\mathcal{T}(c) := \{c' \in \mathcal{C} \mid \text{trip}(c') = \text{trip}(c) \ \wedge\ \tau_{\text{dep}}(c') \geq \tau_{\text{arr}}(c)\}$ be the set of all connections following after c in the trip of c. We then define the PAT for continuing with the same trip as the minimum over the perceived arrival times of all subsequent connections in the trip:

$$\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{trip}) := \begin{cases} \min\{\tau^{\text{p}}(c', d) \mid c' \in \mathcal{T}(c)\} & \text{if } \mathcal{T}(c) \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

Finally, we proceed with defining the PAT $\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{trans})$ for transferring from c to a connection c' of another trip. For this purpose, we first introduce the perceived time $\tau_{\text{trans}}^{\text{p}}(u, v)$ for transferring from u to v as a weighted sum of walking respectively waiting time and transfer costs:

$$\tau_{\text{trans}}^{\text{p}}(u, v) := \begin{cases} \lambda_{\text{trans}} + \lambda_{\text{wait}} \cdot \tau_{\text{trans}}(u, u) & \text{if } u = v \\ \lambda_{\text{trans}} + \lambda_{\text{walk}} \cdot \tau_{\text{trans}}(u, v) & \text{otherwise.} \end{cases}$$

Additionally, we define $\tau_{\text{trans}}^{\text{p}}(c, c') := \tau_{\text{trans}}^{\text{p}}(v_{\text{arr}}(c), v_{\text{dep}}(c'))$, in order to reflect the perceived time for transferring from a connection c to another connection c'. Transferring between connections $c, c' \in \mathcal{C}$ may include some additional waiting time $\tau_{\text{wait}}(c, c')$ at the departure stop of c', after the actual transfer took place. We account for this by introducing the perceived

waiting times $\tau^{\mathrm{p}}_{\mathrm{wait}}(v,\tau,c) := \lambda_{\mathrm{wait}} \cdot \tau_{\mathrm{wait}}(v,\tau,c)$, respectively $\tau^{\mathrm{p}}_{\mathrm{wait}}(c,c') := \lambda_{\mathrm{wait}} \cdot \tau_{\mathrm{wait}}(c,c')$. Using this we define the perceived arrival time $\tau^{\mathrm{p}}_{\mathrm{arr}}(c,c',d) := \tau^{\mathrm{p}}_{\mathrm{trans}}(c,c') + \tau^{\mathrm{p}}_{\mathrm{wait}}(c,c') + \tau^{\mathrm{p}}_{\mathrm{arr}}(c',d)$ of journeys starting with the connection c, followed by a transfer to the connection c', and ending at the destination d. In order to define $\tau^{\mathrm{p}}_{\mathrm{arr}}(c,d \mid \mathrm{trans})$, we only need to specify, which connection c' is used after c. Here, we take not only the perceived arrival time $\tau^{\mathrm{p}}(c',d)$ into account, but also the possibility that the transfer from c to c' might become invalid due to a delay of c. We achieve this by considering all connections that are Pareto-optimal with respect to their PAT and their delay robustness as possible candidates. Based on the set $\mathcal{R}(c) := \{c' \in \mathcal{C} \mid \tau_{\mathrm{wait}}(c,c') \geq 0\}$ of all connections that are reachable from c, the set $\mathcal{R}_{\mathrm{opt}}(c)$ of Pareto-optimal connections, reachable from c can be defined as:

$$\mathcal{R}_{\mathrm{opt}}(c) := \{c' \in \mathcal{R}(c) \mid \forall \bar{c} \in \mathcal{R}(c) : \tau_{\mathrm{wait}}(c,\bar{c}) \geq \tau_{\mathrm{wait}}(c,c') \Rightarrow \tau^{\mathrm{p}}_{\mathrm{arr}}(c,\bar{c},d) \geq \tau^{\mathrm{p}}_{\mathrm{arr}}(c,c',d)\}.$$

Let $\langle c_1,\dots,c_k\rangle$ be the sequence of connections from $\mathcal{R}_{\mathrm{opt}}(c)$ sorted by their waiting time in increasing order, that is $\tau_{\mathrm{wait}}(c,c_i) \geq \tau_{\mathrm{wait}}(c,c_{i-1})$ for $i \in [2,k]$. This means transferring from c to c_1 results in the minimum PAT. If however, transferring to c_1 is not possible, due to delay, c_2 is the next best option, and so on. We define the waiting time when transferring to the i-th connection of the sequence as $\tau^{c}_{\mathrm{wait}}(i) := \tau_{\mathrm{wait}}(c,c_i)$ for $i \in [1,k]$. For $i \neq [1,k]$ we set $\tau^{c}_{\mathrm{wait}}(i) := -\infty$. Finally we define $\tau^{\mathrm{p}}_{\mathrm{trans}}(c,d)$ as the sum of the perceived arrival times of all c_i, weighted by the probability that transfer to c_i is valid, while the transfer to c_{i-1} is invalid:

$$\tau^{\mathrm{p}}_{\mathrm{arr}}(c,d \mid \mathrm{trans}) := \begin{cases} \sum_{i=1}^{k} \left(\frac{P[\tau^{c}_{\mathrm{wait}}(i-1) < \Delta^{c}_{\tau} \leq \tau^{c}_{\mathrm{wait}}(i)]}{P[\Delta^{c}_{\tau} \leq \tau^{c}_{\mathrm{wait}}(k)]} \cdot \tau^{\mathrm{p}}_{\mathrm{arr}}(c,c_i,d) \right) & \text{if } k > 0 \\ \infty & \text{otherwise.} \end{cases}$$

Note that our recursive definition of $\tau^{\mathrm{p}}(c,d)$ is well-defined, since it only depends on the perceived arrival times of connections c' with $\tau_{\mathrm{dep}}(c') > \tau_{\mathrm{dep}}(c)$.

4 Our Approach

Our algorithm is based on a microscopic Monte Carlo simulation of individual passengers represented by a unique integer identifier. For each vertex of the network we maintain a list containing all the passengers, who currently reside at the vertex. Passengers are gradually moved from one vertex to the next, until they reach their destination. We decide for every passenger which vertex he visits next, based on perceived arrival times. The vertex chosen as next vertex is not necessarily the one that minimizes the PAT. We assign a probability based on the perceived arrival times to every possible option. Next, we choose randomly an option for every passenger. In order to increase the accuracy of the simulation we generate λ_{mul} times as many passengers as specified by the demand. After the simulation finished, the results are divided by λ_{mul}, in order to obtain a stochastic distribution of the passengers specified by the demand.

We observe that passengers with the same destination d and roughly the same time of travel, will eventually encounter each other on their journeys (at least at d). If they meet before d, then there exists a vertex v where they have the same options for continuing their journey to d. Our algorithm exploits this observation by evaluating the options and computing the decisions for all passengers at v at once. In order to achieve this, we partition the passengers based on their destination. We proceed with showing how the traffic assignment for passenger with a common destination can be computed. A complete traffic assignment

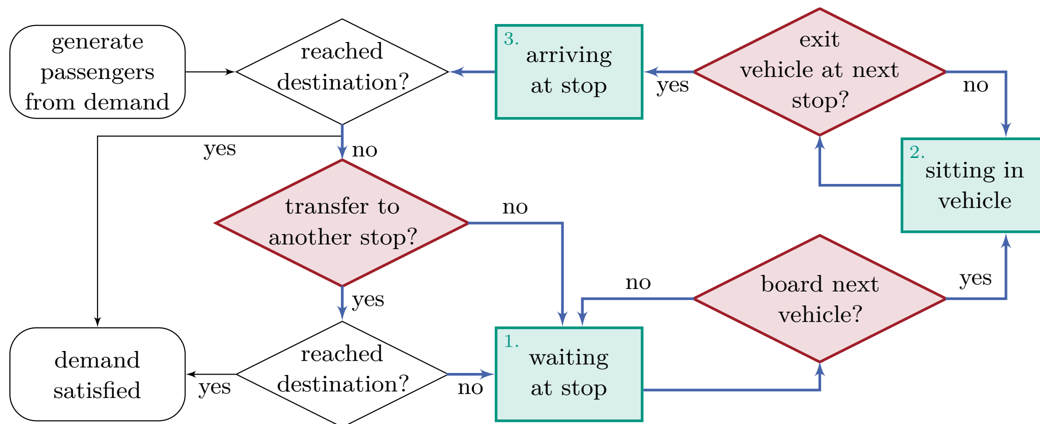


Figure 1 A flowchart describing the movement of a passenger through the network. Passengers are generated according to the demand. If their destination differs from their origin, then they enter the main cycle (colored part) of the simulation. Passengers traverse the main cycle until they reach their destination. During this they can be in one of three situations (green). The situation a passenger is in changes depending on his decisions (red).

can be obtained by doing this for every destination and aggregating the results. For the remainder of this chapter we assume d to be a fixed destination vertex.

We compute the traffic assignment for passengers with destination d in three phases. First, we compute for every connection the minimum perceived arrival times for taking and avoiding that connection. Next, we simulate the movement of the passengers through the network. Every time a passenger could use a connection c without producing an invalid transfer, we decide based on the previously computed perceived transfer times whether the passenger takes the connection c. Connections that are used by the passenger are added to the passengers journey. Finally, we simplify the journeys by removing unwanted cycles.

4.1 Perceived Arrival Time Computation

In the first phase we compute all information required to build journeys one connection at a time. We identified three situations that can occur during the simulation of a passenger's movement, that require a decision about the journey's continuation (see Figure 1).

The first situation arises when a passenger waits at a stop s, while a connection c departs from s. In this case, it has to be decided if the passenger boards the vehicle serving c, or keeps waiting at the stop. In order to make this decision, we need the perceived arrival times for both alternatives. The PAT for using the connection c (i.e. boarding the vehicle) is given by $\tau^{\mathrm{p}}(c,d)$. On the other hand, skipping the connection c and waiting at the stop, implies that some later connection departing from the stop has to be taken. Transferring to another stop is not an option, as the passenger transferred to his current stop s, with the intention to board some vehicle at s. The set of all alternative connections departing from the same stop is given by $\mathcal{A}(c) := \{c' \in \mathcal{C} \mid v_{\mathrm{dep}}(c') = v_{\mathrm{dep}}(c),\ \tau_{\mathrm{dep}}(c') > \tau_{\mathrm{dep}}(c)\}$. We use these alternative connections to obtain the PAT $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathrm{skip}\, c)$ for skipping the connection c as the sum of the additional waiting time and the perceived arrival time of the best alternative connection. Formally, we define: $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathrm{skip}\, c) := \min\{\tau^{\mathrm{p}}_{\mathrm{wait}}(v_{\mathrm{dep}}(c), \tau_{\mathrm{dep}}(c), c') + \tau^{\mathrm{p}}_{\mathrm{arr}}(c', d) \mid c' \in \mathcal{A}(c)\}$.

The second situation affects passengers using a connection that is not the last connection of its trip. These passengers again have to make a binary decision. Either they leave the vehicle at the arrival stop of the current connection, or they use another connection

of the trip. As before, making this decision requires the perceived arrival times of both alternatives. The PAT for continuing with the same trip is given by $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip})$. When disembarking the vehicle, a passenger can continue his journey by either walking to his destination or transferring to another vehicle. Therefore, the PAT for disembarking is given by $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark}) := \min\{\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{walk}),\, \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trans})\}$.

The last situation where a decision has to be made occurs when a passenger leaves a vehicle, but has not yet reached his destination. In this case, it has to be decided to which stop the passenger transfers, in order to wait for another connection. This decision requires a perceived arrival times for every stop v that can be reached by a transfer. Similar to the definition of $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{skip}\, c)$, the PAT for the stop v is given by the PAT of the best connection c departing from v plus the additional waiting time between the arrival time τ at v and the departure of c. As this value is required for every possible arrival time τ at v, we simply compute a profile function $f^{v,d}_{\text{wait}}(\tau)$ for every vertex, which we define as:

$$f^{v,d}_{\text{wait}}(\tau) := \min\{\tau^{\text{p}}_{\text{wait}}(v, \tau, c) + \tau^{\text{p}}_{\text{arr}}(c, d) \mid c \in \mathcal{C},\, \tau_{\text{dep}}(c) \geq \tau,\, v_{\text{dep}}(c) = v\}.$$

In summary, we require for decision making three values per connection: $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip})$, $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{skip}\, c)$, and $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark})$, as well as a profile function $f^{v,d}_{\text{wait}}(\tau)$ per vertex. We now show how these values can be computed in a single sweep over the connection array. As basis for our algorithm, we use the MEAT algorithm [7], which allows for efficient all-to-one profile queries. Instead of computing minimum expected arrival time profiles, as the original MEAT algorithm does, we compute minimum perceived arrival time profiles. In addition to the profile $f^{v,d}_{\text{wait}}(\tau)$, we compute a second profile $f^{v,d}_{\text{trans}}(\tau)$, which we use in order to determine the three PAT values needed per connection. The difference between the two profile is that $f^{v,d}_{\text{trans}}(\tau)$ requires an initial transfer to another stop. Formally, we define:

$$f^{v,d}_{\text{trans}}(\tau) := \min\{\tau^{\text{p}}_{\text{trans}}(v, v_{\text{dep}}(c)) + \tau^{\text{p}}_{\text{wait}}(v, \tau, c) + \tau^{\text{p}}_{\text{arr}}(c, d) \mid c \in \mathcal{C},\, \tau_{\text{wait}}(v, \tau, c) \geq 0\}.$$

Our algorithm maintains for every vertex the two initially incomplete profiles $f^{v,d}_{\text{wait}}(\cdot)$, and $f^{v,d}_{\text{trans}}(\cdot)$. Additionally we store for every trip t a value $\tau^{\text{p}}_{\text{arr}}(t)$, that keeps track of the current value for $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip})$ with $\text{trip}(c) = t$, and is initially ∞. We scan the connection array in decreasing order by departure time. For every connection c we can directly determine the three required values. Since we store the arrival time for continuing with the same trip separately we can set $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip}) \leftarrow \tau^{\text{p}}_{\text{arr}}(\text{trip}(c))$. The PAT for ignoring the connection c is given by the profile that describes waiting at the departure stop of c. Thus, we set $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{skip}\, c) \leftarrow f^{v_{\text{dep}}(c),d}_{\text{wait}}(\tau_{\text{dep}}(c))$. Similarly, the PAT for disembarking at the arrival stop of c is given by the profile that requires an initial transfer. Accordingly, we set $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark}) \leftarrow f^{v_{\text{arr}}(c),d}_{\text{trans}}(\tau_{\text{arr}}(c))$. For the special case that a transfer edge from the arrival stop of c to the destination exists, we have to consider the possibility of walking to the destination. Therefore, we set $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark}) \leftarrow \tau_{\text{arr}}(c) + \tau_{\text{trans}}(v_{\text{arr}}(c), d)$, if this is smaller than the previous value of $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark})$. Afterwards, we temporarily compute the PAT of the connection c: $\tau^{\text{p}}_{\text{arr}}(c, d) \leftarrow \min(\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip}), \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark}))$. We use this value in order to update the profiles and the value $\tau^{\text{p}}_{\text{arr}}(\text{trip}(c))$. First we set $\tau^{\text{p}}_{\text{arr}}(\text{trip}(c)) \leftarrow \tau^{\text{p}}_{\text{arr}}(c, d)$. Next, we add the point $(\tau_{\text{dep}}(c), \tau^{\text{p}}_{\text{arr}}(c, d))$ as a break point to the profile $f^{v_{\text{dep}}(c),d}_{\text{wait}}(\cdot)$, unless this profile already contains a breakpoint with smaller PAT. Finally, we iterate over all vertices v with $(v, v_{\text{dep}}(c)) \in \mathcal{E}$. For each such vertex v we add the point $(\tau_{\text{dep}}(c) - \tau_{\text{trans}}(v, v_{\text{dep}}(c)), \tau^{\text{p}}_{\text{arr}}(c, d))$ as a break point to the profile $f^{v,d}_{\text{trans}}(\cdot)$, unless the profile already contains a breakpoint with smaller PAT. We repeat this process for every connection. Afterwards, we have computed all values required for decision making and can continue with the actual assignment.

4.2 Assignment

The second phase of our algorithm uses the previously computed perceived arrival times to compute the journeys for all the passengers with destination d. To this intent, we maintain for every passenger a list of connections used by the passenger. Additionally, we maintain a list of passengers for every vertex and trip, representing the passengers currently waiting at the vertex, respectively sitting in the vehicle serving the trip. Furthermore, we use a queue sorted by arrival time for every vertex, containing the passengers that are currently transferring to the stop. The transfer queue of each vertex v is initialized with passengers created from the demand with origin v, using their desired departure time as keys.

We now describe how the passengers movement through the network is simulated, using a single scan over the connection array in ascending order by departure time. During this scan, we decide for each connection, which passengers use the connection. When scanning a connection c we first determine the set of passengers that could enter the vehicle. We establish this by removing all the passengers from the transfer queue of $v_{\text{dep}}(c)$ that arrive at $v_{\text{dep}}(c)$ before $\tau_{\text{dep}}(c)$ These passengers are then added to the list of passengers waiting at $v_{\text{dep}}(c)$. Afterwards, the list of passengers waiting at $v_{\text{dep}}(c)$ comprises exactly the passengers that could enter c. We decide whether the passengers take c or not, based on the two PATs $\tau_1 = \min(\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip}), \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark}))$ and $\tau_2 = \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{skip}\, c)$. We do so by assigning a probability $P[i]$, that describes the likelihood of a passenger using the option associated with τ_i, to each of the alternatives.

In general, given k options, with perceived arrival times $\tau_1, \ldots, \tau_k$, we define the probability $P[i]$ for choosing option i as follows. First, we compute the *gain* of each option, which we define as $g(i) := \max(0, \min_{j \neq i}(\tau^{\text{p}}(j)) - \tau^{\text{p}}(i) + \lambda_{\Delta\text{max}})$. Doing so results in a gain of zero, for options that differ from the optimum by more than $\lambda_{\Delta\text{max}}$. For all other options τ_i, the gain correlates linearly to the difference between τ_i and the optimal, respectively nest best option. The probability that a passenger uses option i is equivalent to the gain of option i, divided by the sum of the gain of all other options. Formally we define: $P[i] := g(i)/\sum_{j=1}^{k} g(j))$.

Using this, the probabilities of the two options τ_1 (for using the connection c), and τ_2 (for skipping the connection), are given by $P[1] := (\tau_2 - \tau_1 + \lambda_{\Delta\text{max}})/2\lambda_{\Delta\text{max}}$, and $P[1] := 1 - P[2]$. Based on these probabilities, we make a random decision for every passenger waiting at the departure stop of c. If a passenger happens to enter the connection, then he is removed from the list of passengers waiting at $v_{\text{dep}}(c)$, and added to the list of passengers sitting the trip trip(c). Furthermore, the connection c is added to the journey of the passenger.

Next, we decide for every passenger sitting in the trip, if he disembarks at the arrival stop of the connection. In this case, the two options are given by $\tau_1 = \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark})$ for leaving the vehicle, and $\tau_2 = \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip})$ for continuing with the same trip. As before we compute the probabilities of both options, and make a random decision for every passenger sitting in the trip, based on these probabilities. Passengers disembarking the vehicle are collected in a temporary list. If the arrival stop of the connections happens to be the destination vertex, then the journeys of all passengers in the temporary list are complete, and we simply continue with the next connection. Otherwise, we have to decide for all passenger in the temporary list, to which vertex they transfers. Let $v_1, \ldots, v_k$ be all vertices for which $\tau_{\text{trans}}(v_{\text{arr}}(c), v_i) < \infty$ holds. The perceived arrival time for transferring to v_i is given by $\tau_i = \tau^{\text{p}}_{\text{trans}}(v_{\text{arr}}(c), v_i) + f^{v_i,d}_{\text{wait}}(\tau_{\text{arr}}(c) + \tau_{\text{trans}}(v_{\text{arr}}(c), v_i))$. Based on these perceived arrival times, we compute the probability of a passenger transferring to vertex v_i, for $i \in [1, k]$. As before, we determine for every passenger randomly, which option he chooses. Finally, passengers transferring to vertex v are added to the queue of transferring passengers of the vertex v, their arrival time at v is $\tau_{\text{arr}}(c) + \tau_{\text{trans}}(v_{\text{arr}}(c), v)$. We repeat this process

for every connection$c \in \mathcal{C}$. After processing every connection, we have assigned journeys to all passengers except the ones where no valid journey exists.

4.3 Cycle Elimination

During the second phase, we assigned a journey to every passenger which might not necessarily be an optimal journey. Therefore it is possible that the assigned journey contains cycles. In fact, it is even possible that a journey that is optimal with respect to perceived arrival time can contain cycles. This could be the case if the waiting cost λ_{wait} is very high, such that driving in a circle instead of waiting reduces the perceived arrival time. However, for some applications it might be undesirable or inadmissible to assign journeys containing cycles. Thus, we now describe an optional third phase of our algorithm, that removes all cycles from the assigned journeys.

In order to detect and remove cycles from a journey $j = \langle c_1, \ldots, c_k \rangle$ satisfying a demand D, we first convert it into a sequence $\langle (v_1, \tau_1), \ldots, (v_{2k+2}, \tau_{2k+2}) \rangle$ of vertex, time pairs. We do so by setting $v_{2i} := v_{\text{dep}}(c_i)$, $\tau_{2i} := \tau_{\text{dep}}(c_i)$, $v_{2i+1} := v_{\text{arr}}(c_i)$, and $\tau_{2i+1} := \tau_{\text{arr}}(c_i)$, for $i \in [1, k]$. Furthermore we define the first and last pair as $v_1 := o(D)$, $\tau_1 := \tau_{\text{dep}}(D)$, $v_{2k+2} := d(D)$, and $\tau_{2k+2} := \infty$. Given this, we say that the journey contains a cycle, if their exist indices i and $j > i + 1$, such that the part of the journey between vertices v_i and v_j can be replaced by a transfer. This is possible if $\tau_i + \tau_{\text{trans}}(v_i, v_j) \leq \tau_j$ holds. Since the transfer graph is transitively closed, it consist of disjoint cliques. Thus a journey can only contain a cycle if it contains two vertices v_i, v_j of the same clique. We can check this efficiently while iterating through the sequence of vertex, time pairs. For every $i \in [1, 2k + 2]$ add i to a set associated with the clique that contains v_i Afterwards we check for each of these sets, if it contains indices i, j such that $\tau_i + \tau_{\text{trans}}(v_i, v_j) \leq \tau_j$ holds. If we found such indices i, j, then we have also found a cycle that can be replaced with a transfer. We remove this cycle be removing the connections $c_{\lfloor i/2 \rfloor}, \ldots, c_{\lceil j/2 \rceil}$ from the journey.

4.4 Parallelization

Our algorithm begins with a short setup phase, during which the connections get sorted, and the passengers get divided by their destination. Afterwards, a separate assignment is computed for every destination. Finally, the results are aggregated and the algorithm terminates. The assignment computation for the different destinations is by far the most complex part of the algorithm and can be performed for every destination independently. Therefore, it is quite easy to parallelize this part of the algorithm. First, the destinations a distributed among the available processors. Afterwards each processor computes an independent assignment for the corresponding destinations.

5 Evaluation

We implemented our algorithm in C++ compiled with GCC version 5.3.1 and optimization flag -O3. Experiments were conducted on a quad core Intel Xeon E5-1630v3 clocked at 3.7 GHz, with 128 GiB of DDR4-2133 RAM, 10 MiB of L3 cach, and 256 KiB of L2 cache.

5.1 Instance

We tested our algorithm on a public transit network covering the greater region of Stuttgart, as well as some long distance routes, reaching as far as Mannheim, Basel or Munich. This

Table 1 Instance size.

#Vertices	15 115
#Stops	13 941
#Edges	33 890
#Edges − #Loops	18 775
#Connections	780 042
#Trips	47 844
#Passenger	1 249 910

Table 2 Running time of our algorithm depending on the maximum delay $\Delta_\tau^{\max}$.

$\Delta_\tau^{\max}$	time
1 min	108.57 sec
2 min	109.92 sec
4 min	111.49 sec
8 min	117.32 sec
16 min	125.26 sec
32 min	136.25 sec
64 min	149.61 sec

network as well as the associated model for demand was introduced in [9]. The timetable covers roughly the traffic of one day, the earliest connection departs at 0:39 am and the last connection arrives at 2:37 am on the second day. Some key figures of the network are listed in Table 1. Since we require the transfer graph to be transitively closed, it contains a loop edge at every vertex, which increases the number of total edges significantly. Our algorithm depends on several tuning parameters that are used to adequately model passenger behavior. For our experiments, we chose the following values: the walking cost is set to $\lambda_{\text{walk}} = 2.0$, the waiting cost is set to $\lambda_{\text{wait}} = 0.5$, the transfer cost as well as the delay tolerance are set to $\lambda_{\text{trans}} = \lambda_{\Delta\max} = 300$ sec, and the maximum delay is set to $\Delta_\tau^{\max} = 60$ sec.

5.2 Experiments

Our first experiment evaluates the performance of our algorithm, depending on the various tuning parameters. These parameters are primarily intended to model different passenger preferences. As such they do not directly influence the computational complexity of the algorithm. In fact there is no measurable difference in running times when the parameters λ_{walk}, λ_{wait}, λ_{trans}, or $\lambda_{\Delta\max}$ are changed. However, increasing the maximum delay $\Delta_\tau^{\max}$ of the connections slightly increases the running time, as stated in Table 2. For every row in the table we repeated the assignment computation ten times and report the mean of the resulting running times. The increase in running time is caused by the computation of $\tau_{\text{arr}}^{\text{p}}(c, d \mid \text{trans})$, since more connections have a non zero probability of being the successor connection for c.

Another important tuning parameter, is the passenger multiplier λ_{mul}. Changing λ_{mul} directly influences the amount of work that has to be done, since more passengers have to be simulated. Figure 2 shows the running time of our algorithm dependent on λ_{mul}, differentiated by the phases of the algorithm. As expected the running time increases with an increasing passenger multiplier. The additional running time is mostly due to the assignment

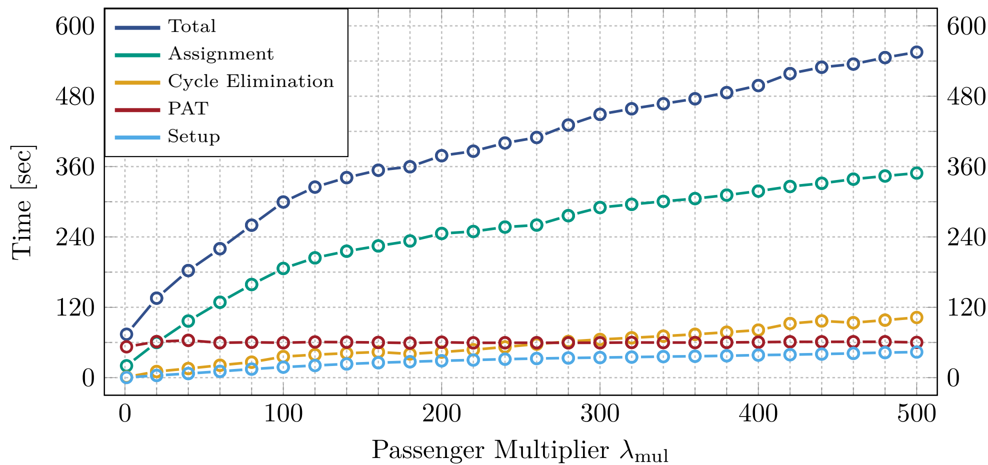


Figure 2 The running time of our algorithm depending on the passenger multiplier, differentiated by the phases of the algorithm. Changing the passenger multiplier primarily affects the assignment phase. Every measurement is the mean over the running times of ten repetitions of our algorithm.

phase. However, the running time of the assignment phase is not doubled when the number of passengers is doubled. This is the case, because an increased number of passengers leads to more passengers making the same decisions. Thus synergy effects can be used during the computation. The PAT computation is completely independent from the number of passengers, which leads to a constant running time as seen in Figure 2. The time required for the cycle elimination and the setup phase (i.e. sorting the connections and distributing the passengers by destination) increases only slightly with an increased passenger multiplier.

Next, we evaluate the performance of the parallelized version of our algorithm. For the following experiment we use a passenger multiplier of $\lambda_{\text{mul}} = 10$, since this is in most cases sufficient for an accurate result. The serial version of the algorithm has a running time of 108.57 sec. Using the parallelized version with only one thread results in a slightly increased running time of 108.92 sec. Using two threads we achieve a running time of 65.57 sec, four threads achieve 38.41 sec. As before all measurements are the mean over ten executions. Using four threads we only achieve a speed-up of 2.83, despite the fact that the computations are complete independent of each other. This could be the case, because our algorithm primarily scans through the memory. Thus, memory bandwidth could be a limiting factor.

Additionally, we compare the running time of our algorithm to VISUM, which is a commercial tool from PTV AG. On the same instance the VISUM computation took just above 30 minutes, and was parallelized using 8 threads. The VISUM assignment was computed on an Intel Core i7-6700 clocked at 3.4 GHz with 64 GiB of RAM, running Windows 10. Thus our algorithm outperforms the state-of-the-art by a factor of about 50.

Finally, we compare the quality of the assignment computed by our algorithm to the one computed by VISUM. Table 3 summarizes the results. Overall, the assignments computed by our algorithm and VISUM are quite similar. Our algorithm assigns journeys with slightly longer mean travel time, in favor of a slightly decreased number of transfers. At the same time, our algorithm assigns journeys with a higher maximum number of trips. The reason for this is that VISUM prunes all journeys with more than 6 trips, while our algorithm has no hard limit on the number of transfers. It is noticeable, that both techniques assign about 1200 passengers to a single vehicle, since both are not able to handle vehicle capacities.

Table 3 Comparison between an assignment computed by VISUM and our algorithm. We report for every quantity the minimum (min), mean, standard deviation (sd), and maximum (max) over all journeys. The figures for both assignments are quite similar. However, our assignment slightly favors journeys with fewer trips (transfers), at the disadvantage of marginal increased travel time.

	VISUM				Our Algorithm			
Quantity	min	mean	sd	max	min	mean	sd	max
Total travel time [min]	2.98	46.885	23.753	429.00	2.98	47.199	23.443	429.00
Time spent in vehicle [min]	0.02	21.059	18.796	380.00	0.02	21.231	18.749	323.97
Time spent walking [min]	2.00	22.394	5.200	149.00	2.00	22.476	5.265	149.00
Time spent waiting [min]	0.00	3.432	5.722	217.02	0.00	3.492	5.677	217.02
Trips per passenger	1.00	1.771	0.833	6.00	1.00	1.746	0.843	8.00
Connections per passenger	1.00	9.396	7.435	109.00	1.00	9.474	7.331	97.00
Passengers per connection	0.00	12.740	37.795	1 290.10	0.00	12.847	37.584	1 233.60

6 Conclusion and Future Work

In this work we presented a novel algorithmic approach to compute public transit traffic assignments. As a means of modeling realistic passenger behavior we introduced perceived arrival times. This allowed us to consider several important criteria of a journey while developing an efficient algorithm. We showed that the resulting assignment is comparable to the state-of-the-art in terms of quality. Concerning running time, our algorithm is more than an order of magnitude faster then state-of-the-art.

For future work, it would be interesting to incorporate vehicle capacities and an equilibrium model in our approach, since our experiments showed that a lack of those results in some vehicles having a very high utilization. Moreover, we would like to incorporate the cycle elimination phase into the assignment phase, such that journeys containing cycles are not assigned in the first place.

References

1. Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.
2. Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In *Algorithm Engineering – Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.
3. Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-robustness of transfer patterns in public transportation route planning. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*, Open-Access Series in Informatics (OASIcs), pages 42–54, 2013. `doi:10.4230/OASIcs.ATMOS.2013.42`.
4. Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann. Accelerating time-dependent multi-criteria timetable information is harder than expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling,*

Optimization, and Systems (ATMOS'09), OpenAccess Series in Informatics (OASIcs), 2009. doi:10.4230/OASIcs.ATMOS.2009.2148.

5. Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. *Transportation Science*, 2014. doi:10.1287/trsc.2014.0534.
6. Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2013.
7. Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Delay-robust journeys in timetable networks with minimum expected arrival time. In *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*, OpenAccess Series in Informatics (OASIcs), 2014. doi:10.4230/OASIcs.ATMOS.2014.1.
8. Michael Patriksson. *The Traffic Assignment Problem: Models and Methods.* Courier Dover Publications, 2015.
9. Johannes Schlaich, Udo Heidl, and Regine Pohlner. Verkehrsmodellierung für die Region Stuttgart – Schlussbericht. Unpublished manuscript, 2011.
10. Yosef Sheffi. *Urban Transportation Networks.* Prentice-Hall, Englewood Cliffs, NJ, 1985.
11. Ben Strasser and Dorothea Wagner. Connection scan accelerated. In *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX'14)*, pages 125–137. SIAM, 2014.
12. Sascha Witt. Trip-based public transit routing. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, Lecture Notes in Computer Science. Springer, 2015. Accepted for publication.

Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure

Tobias Heuer[1] and Sebastian Schlag[2]

1 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`tobias.heuer@gmx.net`

2 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`sebastian.schlag@kit.edu`

Abstract

We present an improved coarsening process for multilevel hypergraph partitioning that incorporates global information about the community structure. Community detection is performed via modularity maximization on a bipartite graph representation. The approach is made suitable for different classes of hypergraphs by defining weights for the graph edges that express structural properties of the hypergraph. We integrate our approach into a leading multilevel hypergraph partitioner with strong local search algorithms and perform extensive experiments on a large benchmark set of hypergraphs stemming from application areas such as VLSI design, SAT solving, and scientific computing. Our results indicate that respecting community structure during coarsening not only significantly improves the solutions found by the initial partitioning algorithm, but also consistently improves overall solution quality.

1998 ACM Subject Classification G.2.2 Graph Theory, G.2.3 Applications

Keywords and phrases multilevel hypergraph partitioning, coarsening algorithms, community detection

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.21

1 Introduction

Hypergraphs are a generalization of graphs, where each (hyper)edge (also called *net*) can connect more than two vertices. The k-way hypergraph partitioning problem is the generalization of the well-known graph partitioning problem: partition the vertex set into k disjoint blocks of bounded size (at most $1 + \varepsilon$ times the average block size), while minimizing an objective function defined on the nets. Hypergraph partitioning (HGP) has a wide range of applications. Two prominent areas are VLSI design and scientific computing (e. g. accelerating sparse matrix-vector multiplications) [53]. While the former is an example of a field where small optimizations can lead to significant savings [63], the latter exemplifies problems where hypergraph-based modeling is more flexible than graph-based approaches [16, 34, 35, 36, 43]. HGP also finds application as a preprocessing step in SAT solving, where it is used to identify groups of connected variables [3, 24, 48].

Since hypergraph partitioning is NP-hard [46] and since it is even NP-hard to find good approximate solutions for graphs [14], heuristic *multilevel* algorithms [15, 19, 33, 37] are used in practice. These algorithms consist of three phases: In the *coarsening phase*, the hypergraph is coarsened to obtain a hierarchy of smaller hypergraphs. After applying an *initial partitioning* algorithm to the smallest hypergraph in the second phase, coarsening is undone and, at each level, a *local search* method is used to improve the partition induced by the coarser level.

© Tobias Heuer and Sebastian Schlag;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 21; pp. 21:1–21:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Coarsening is deemed to be the most important phase of the multilevel paradigm and an area where future research is required to devise algorithms that are suitable for a wide range of hypergraphs [40]. In order to create hypergraphs that are *smaller than* but structurally *similar to* the given hypergraph, coarsening schemes try to identify and merge naturally existing clusters of vertices [22]. State of the art hypergraph partitioning tools compute matchings or clusterings using *local* similarity measures that only take into account the direct neighborhood of each vertex [8, 5, 16, 22, 41, 42, 59, 60, 61]. *Global* considerations are avoided due to the high running times of the respective algorithms [58].

Outline and Contribution. After introducing basic concepts and summarizing related work in Section 2, we present our community-aware hypergraph coarsening framework in Section 3. In a preprocessing phase, we perform modularity maximization on a bipartite graph representation to detect *global* community structure in the input hypergraph. This information is used to guide the coarsening process and to prevent contractions that obscure naturally existing clustering structure. By incorporating information about the net sizes and vertex degrees into the edge weights of the bipartite graph, we make our approach suitable for different classes of hypergraphs. We implemented our algorithm in the open source HGP framework KaHyPar [1]. Extensive experiments presented in Section 4 indicate that respecting community structure during coarsening significantly improves solution quality while having only a moderate impact on the running time. Section 5 concludes the paper.

2 Preliminaries

Notation and Definitions. An *undirected hypergraph* $H = (V, E, c, \omega)$ is defined as a set of n vertices V and a set of m hyperedges/nets E with vertex weights $c : V \to \mathbb{R}_{>0}$ and net weights $\omega : E \to \mathbb{R}_{>0}$, where each net is a subset of the vertex set V (i.e., $e \subseteq V$). The vertices of a net are called *pins*. We use P to denote the multiset of all pins in H. We extend c and ω to sets, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e if $v \in e$. $\mathrm{I}(v)$ denotes the set of all incident nets of v. The *degree* of a vertex v is $d(v) := |\mathrm{I}(v)|$. The set $\Gamma(v) := \{u \mid \exists\, e \in E : \{v, u\} \subseteq e\}$ denotes the neighbors of v. The *size* $|e|$ of a net e is the number of its pins. Nets of size one are called *single-vertex* nets. A *k-way partition* of a hypergraph H is a partition of its vertex set into k *blocks* $\Pi = \{V_1, \dots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. We call a k-way partition Π *ε-balanced* if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some parameter ε. Given a k-way partition Π, the number of pins of a net e in block V_i is defined as $\Phi(e, V_i) := |\{v \in V_i \mid v \in e\}|$. For each net e, $\Lambda(e) := \{V_i \mid \Phi(e, V_i) > 0\}$ denotes the *connectivity set* of e. The *connectivity* of a net e is the cardinality of its connectivity set: $\lambda(e) := |\Lambda(e)|$. A net is called *cut net* if $\lambda(e) > 1$. The *k-way hypergraph partitioning problem* is to find an ε-balanced k-way partition Π of a hypergraph H that minimizes an objective function over the cut nets for some ε. Several objective functions exist in the literature [6, 46]. The most commonly used cost functions are the *cut-net* metric $\mathrm{cut}(\Pi) := \sum_{e \in E'} \omega(e)$ and the *connectivity* metric $(\lambda - 1)(\Pi) := \sum_{e \in E'} (\lambda(e) - 1)\ \omega(e)$, where E' is the set of all cut nets [23]. In this paper, we use the connectivity-metric, which accurately models the total communication volume of parallel sparse matrix-vector multiplication [16]. Optimizing both objective functions is known to be NP-hard [46]. *Contracting* a pair of vertices (u, v) means merging v into u. The weight of u becomes $c(u) := c(u) + c(v)$. We connect u to the former neighbors $\Gamma(v)$ of v by replacing v with u in all nets $e \in \mathrm{I}(v) \setminus \mathrm{I}(u)$ and remove v from all nets $e \in \mathrm{I}(u) \cap \mathrm{I}(v)$. *Uncontracting* a vertex u reverses the contraction. The two most common

ways to represent a hypergraph $H = (V, E, c, \omega)$ as an undirected graph are the *clique* and the *bipartite* representation [38]. In the following, we use *nodes* and *edges* when referring to a graph representation and *vertices* and *nets* when referring to H. In the *clique* graph $G_x(V, E_x \subseteq V^2)$ of H, each net is replaced with an edge for each pair of vertices in the net: $E_x := \{(u, v) : u, v \in e, e \in E\}$. Thus the pins of a net e with size $|e|$ form a $|e|$-clique in G_x. In the *bipartite* graph $G_*(V \dot\cup E, F)$ the vertices and nets of H form the node set and for each net e incident to a vertex v, we add an edge (e, v) to G_*. The edge set F is thus defined as $F := \{(e, v) \mid e \in E, v \in e\}$. Each net in E therefore corresponds to a star in G_*. In both models, node weights c and edge weights ω are chosen according to the problem domain [31].

Related Work. Since the 1990s HGP has evolved into a broad research area. We refer to [6, 9, 53, 58] for an extensive overview, and focus instead on issues closely related to the contributions of our paper. Well-known multilevel HGP software packages with certain distinguishing characteristics include PaToH [16] (originating from scientific computing), hMetis [41, 42] (originating from VLSI design), Mondriaan [61] (sparse matrix partitioning), MLPart [5] (circuit partitioning), Zoltan [22] and Parkway [59] (parallel), UMPa [60] (directed hypergraph model, multi-objective), and kPaToH (multiple constraints, fixed vertices) [8]. All of these algorithms compute vertex matchings [5, 8, 16, 22, 61] or clusterings [16, 41, 42, 59] on each level of the coarsening hierarchy. Different rating functions are used to determine the vertices to be matched or clustered together. All clustering algorithms proceed in a local and greedy fashion: For each vertex the neighbor that maximizes the rating function is chosen as contraction partner. Global decisions are avoided due to the high running times of the respective algorithms [58]. Hagen and Kahng [32] propose a $\mathcal{O}(n^3)$ time algorithm that uses cycles in random walks of the clique representation to identify global clustering structure. Cong and Lim [18] use approximate edge separability computations as a global clustering measure and give an algorithm that runs in $\mathcal{O}(m + n \log n)$ time on the clique representation with m edges and n nodes. Note that for sparse hypergraphs the number of edges in the clique representation can be as high as $m \in \mathcal{O}(n^2)$. Lotfifar and Johnson [47] suggest to cluster hyperedges and to remove less important ones to make better global vertex clustering decisions using rough set clustering.

KaHyPar. The **Ka**rlsruhe **Hy**pergraph **Par**titioning framework instantiates the multilevel approach in its most extreme version, removing only a single vertex in every level of the hierarchy. By using this very fine grained n-level approach combined with strong local search heuristics, KaHyPar seems to be the method of choice for optimizing the cut- and the $(\lambda - 1)$-metric unless speed is more important than quality [1, 56]. Currently, it contains two coarsening algorithms. The first algorithm [56] starts with calculating the locally best contraction partner $u \in \Gamma(v)$ for each vertex v according to a rating function. Then the contractions are performed in decreasing rating score order. Ratings are stored in a priority queue and kept up-to-date during the coarsening process. Thus the algorithm always contracts the vertex pair with the globally highest rating. In the second algorithm vertices are visited in *random* order and each vertex is immediately contracted with its highest-rated neighbor. This approach is shown to not affect the solution quality, while being significantly faster than the first algorithm [1].

Community Detection via Modularity Maximization. Community detection tries to extract an underlying structure from a graph by dividing its nodes into disjoint subgraphs (communities) such that connections are dense *within* subgraphs but sparse between them [28, 55].

Different quality functions are used to judge the goodness of a division into communities. The most popular quality function is the *modularity* of Newman and Girvan [52], which compares the observed fraction of edges within a community with the expected fraction of edges if edges were placed using a random edge distribution that preserves the degree distribution of the graph [27]. More formally, given a graph G and disjoint communities $C = \{C_1, \dots, C_x\}$, modularity is defined as:

$$Q := \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(C_i, C_j) \tag{1}$$

where A_{ij} is the entry of the adjacency matrix A representing edge (i, j), $m = \frac{1}{2} \sum_{ij} A_{ij}$ is the number of edges in the graph, k_i is the degree of node i, C_i is the community of vertex i, and δ is the Kronecker delta. Note that this can be generalized to weighted graphs: A_{ij} represents the weight of edge (i, j), $k_i = \sum_j A_{ij}$ is the weighted degree of node i and $m = \frac{1}{2} \sum_{ij} A_{ij}$ is the sum of all edge weights [51]. Modularity optimization is known to be NP-hard [13], but several efficient heuristics exist. A fast and widely used algorithm is the Louvain method introduced by Blondel et al. [12]: Initially, each node is assigned to a community of its own. Then the algorithm proceeds in two phases that are repeated iteratively. In the first phase, nodes are repeatedly assigned to the neighboring community that maximizes the increase in modularity. This local, greedy optimization stops when no further increase is possible. In the second phase, the graph is coarsened according to the community structure discovered in the first phase by contracting each community into a single node. Then, the process starts again on the coarsened graph and is repeated until the maximum modularity is achieved. The communities of the coarsest graph determine the community structure of the input graph. The algorithm has low computational complexity and is thus suitable for large graphs [28, 45]. There exist several definitions of modularity adapted specifically to bipartite graphs [10, 30, 39, 49]. However, we do not consider these in this work, since they do not translate into fast algorithms and therefore only scale to small bipartite graphs [49]. We note that there also exist techniques to detect communities in k-partite, k-uniform hypergraphs. In these approaches, hypergraphs are projected to k bipartite graphs and bipartite modularity measures are used to detect community structures [50].

3 Community-aware Coarsening

There are three main design goals underlying coarsening schemes of multilevel hypergraph partitioning algorithms [42]:

1. Coarsening should successively reduce the size of the nets, because small nets allow move-based local search algorithms to identify moves that improve the solution quality more easily.
2. Coarsening should successively reduce the number of nets in the coarser hypergraphs. This can be accomplished by preferring contractions that create single-vertex nets and leads to simpler instances for initial partitioning, since single-vertex nets cannot be cut.
3. Vertices should be contracted in such a way that the initial partitioning algorithm is able to compute a high-quality solution, i.e. the partition of the coarsest level should not be significantly worse than the final partition of the hypergraph. Therefore, it is necessary that the coarse approximations remain *structurally similar* to the input hypergraph.

To accommodate goals one and two, state-of-the-art HGP libraries use rating functions to identify and contract highly connected vertices such that the number of nets and their size is successively reduced. The most commonly used rating function is *heavy-edge*: Given

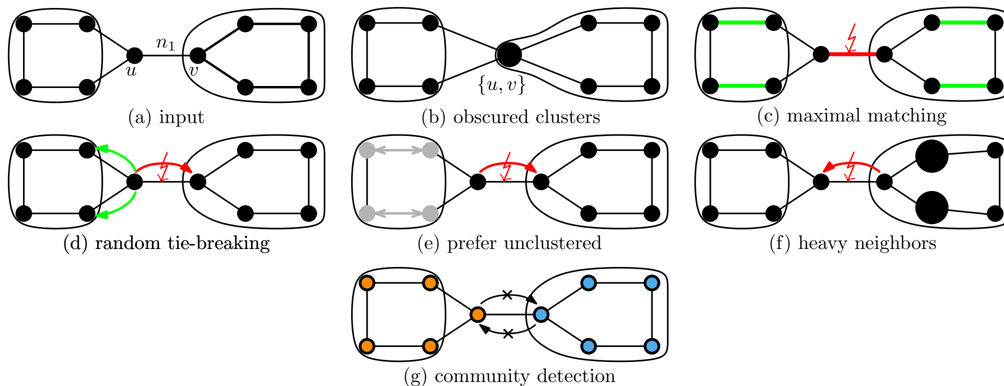


Figure 1 (a) Hypergraph with 10 vertices and 13 nets. Nets containing only two vertices are shown as graph edges. By cutting net n_1 the hypergraph can be partitioned into two balanced blocks. (b) Contracting vertex pair (u, v) obscures the naturally existing clustering structure and the cut of size 1. (c)–(f) Properties of coarsening algorithms that lead to the contraction of (u, v): (c) Coarsening is based on maximal matchings. (d) Random tie-breaking among all neighbors with same rating score. (e) Preferring unclustered vertices to break ties. (f) Contraction partners with highest rating score are already too heavy. (g) Our approach: Restrict contractions to vertex pairs *within* the same community. This prevents the contraction of (u, v) in all aforementioned cases.

two vertices u and $v \in \Gamma(u)$, it is defined as $r(u, v) := \sum_{e \in E'} \omega(e)/(|e| - 1)$, where $E' := \{\mathrm{I}(v) \cap \mathrm{I}(u)\}$. This rating is employed in several tools including hMetis [41], Parkway [59], KaHyPar [1, 56] and PaToH [17] and prefers vertex pairs that share a large number of heavy nets with small size. *Structural similarity* between the coarser approximations and the original hypergraph (goal three) is maintained by allowing the formation of vertex clusters instead of enforcing matchings, since their maximality constraint can destroy some naturally existing clusters in the hypergraph [40] (see Fig. 1 (a)–(c) for an example). Furthermore the algorithms ensure that the distribution of vertex weights does not become too imbalanced at the coarsest level, since this limits the number of feasible initial partitions satisfying the balance constraint. This is done by either enforcing an upper-bound on the vertex weight or by integrating a penalty factor into the rating function that discourages the formation of heavy vertices.

However, since coarsening decisions are only based on *local* information, several situations can arise in which naturally existing structure is obscured: If multiple neighbors have the same rating score, coarsening algorithms employ different tie-breaking strategies such as randomly choosing one of them or giving preference to vertices that have not yet been clustered [1, 40] (see Fig. 1 (d),(e)). Furthermore, a restriction on the maximum allowed vertex weight can lead to situations in which the highest rated contractions are forbidden by the weight constraint. Therefore the coarsening algorithm performs a contraction with lower rating score (Figure 1 (f)). Situations like these arise, because all coarsening algorithms are guided by local, greedy decisions based on rating functions that solely consider the weights and sizes of nets connecting candidate vertices and therefore lack a global view of the clustering problem. If information about the community structure were to be known before the coarsening process, these cases could have been prevented explicitly. We therefore propose an approach to combine a *global* view on the problem with *local* coarsening decisions.

Community-aware Coarsening Framework. Our framework consists of two phases. First, a (graph-based) community detection algorithm is used to partition the vertices of the hyper-

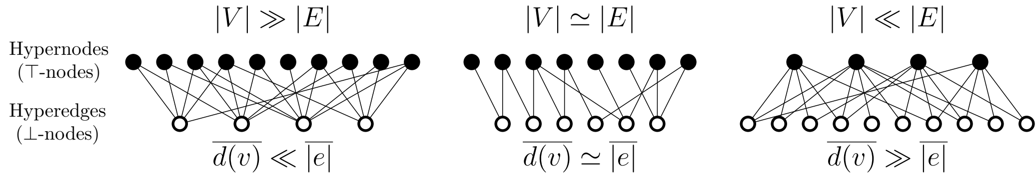


Figure 2 Bipartite graph-based representations of hypergraphs of varying density. In hypergraphs with low density, the bipartite graph consists of many $\top$-nodes with low average degree and fewer $\bot$-nodes with high average degree (left). If $d \approx 1$, the number of $\top$- and $\bot$-nodes and their average degrees are roughly equal (middle). High-density hypergraphs lead to bipartite representations with fewer $\top$-nodes with high average degree and many $\bot$-nodes with low average degree (right).

graph into a set $C = \{C_1, \ldots, C_x\}$ of internally densely and externally sparsely connected communities. The actual number of communities $|C|$ is determined by the community detection algorithm. Then, a hypergraph coarsening algorithm is applied on each community C_i independently. This can be accomplished by modifying the algorithm to only contract vertices within the *same* community, i.e. given a vertex $u \in C_i$, we restrict potential contraction partners to $\Gamma(u) \cap C_i$. By preventing inter-community contractions, the coarsening algorithm maintains the structural similarity discovered by the community detection algorithm, while still allowing local, intra-community decisions to be based on rating functions tailored to the HGP problem. Note that this framework is *independent* of the algorithms used for community detection and coarsening. In the following, we describe one instantiation, which performs community detection via modularity maximization using the Louvain method.

Hypergraph Representation. In order to employ the Louvain method as community detection algorithm, a suitable graph-based representation of the hypergraph has to be chosen. As described in Section 2 the two common models are the clique and the bipartite representation. However, several reasons make the clique representation unsuitable for our purpose. Inserting $\binom{|e|}{2}$ graph edges into the clique graph for every net e destroys the natural sparsity of the hypergraph [6] and therefore may be prohibitively costly in terms of both space and running time. Furthermore and more importantly, this exaggerates the importance of nets with more than two pins [57], since large nets automatically imply a high edge density in the clique representation. We therefore use the *bipartite* representation, which allows us to encode any hypergraph in $\mathcal{O}(|P|)$ space. In the following, we refer to the graph nodes representing the vertices of the hypergraph as $\top$-nodes and to the nodes representing the nets as $\bot$-nodes (see Figure 2 for an example).

Modeling Peculiarities. By performing community detection on the bipartite graph representation we receive a community partition of *both* the vertices *and* the nets of the hypergraph, since both are represented as $(\top, \bot)$-nodes in the graph. However, we are only interested in the community structure of the vertices. Therefore we have to take structural properties of the hypergraphs into account. More specifically, we have to consider the density:

$$d := \frac{\overline{d(v)}}{\overline{|e|}} = \frac{|P|/n}{|P|/m} = \frac{m}{n}, \tag{2}$$

where $\overline{d(v)}$ is the average vertex degree and $\overline{|e|}$ is the average net size. If $d \approx 1$, the number of $\top$-nodes is roughly equal to the number of $\bot$-nodes and $\overline{d(v)} \simeq \overline{|e|}$. If $d \gg 1$ then there are more $\bot$-nodes than $\top$-nodes and $\overline{d(v)} \gg \overline{|e|}$, whereas if $d \ll 1$ the opposite is the case (see Figure 2). In case the hypergraph exhibits low density and therefore a large average net size,

special care has to be taken in order to ensure that the community structure is not exclusively shaped by the high-degree $\bot$-nodes. Similarly, the large number of $\bot$-nodes can lead to a community structure that is dominated by the nets of the hypergraph in the high-density case. Hypergraphs with density $d \approx 1$ do not pose a problem, since the number of $\top$-nodes and $\bot$-nodes as well as their degrees are balanced. We account for these structural differences by encoding additional information about the hypergraph structure into the weights of the bipartite graph edges.

Weighting Graph Edges. We propose three different weights for the edges (v, e) between $\top$-nodes $v \in V$ and $\bot$-nodes $e \in E$ as shown in Eq. 3. The first scheme uses uniform edge weights as a baseline. Giving each edge an equal weight is expected to provide good clustering results for hypergraphs with $d \approx 1$, since for these instances the number of $\top$- and $\bot$-nodes as well as their degrees are roughly comparable. The second and third schemes account for the skew in low and high density hypergraphs. The weighting function ω_e assigns each edge a weight which is inversely proportional to the size of the net, i.e. smaller nets get a higher influence on the community structure than larger nets. If many small nets are contained within a community, the coarsening algorithm can successively reduce their size and eventually remove them from the hypergraph (goals one and two). Furthermore, this ensures that high-degree $\bot$-nodes (i.e., large nets) do not dominate the community structure by attracting to many $\top$-nodes. This edge weight only affects the clustering decisions of $\top$-nodes, since from the perspective of $\bot$-nodes each outgoing edge still has uniform weight $1/|e|$. In order to also influence the clustering decision of $\bot$-nodes, the third weighting function ω_{de} additionally integrates the hypernode degree into the edge weight. Strengthening the connection between $\bot$-nodes and high-degree $\top$-nodes facilitates the formation of communities around high-degree vertices in the hypergraph. Note that it is possible to efficiently choose an appropriate weighting scheme at runtime by calculating the density of the hypergraph according to Eq. 2 and modifying the edge weights appropriately.

$$\omega(v,e) := 1 \qquad \omega_{\mathrm{e}}(v,e) := \frac{1}{|e|} \qquad \omega_{\mathrm{de}}(v,e) := \frac{d(v)}{|e|} \tag{3}$$

4 Experimental Evaluation

We implemented modularity-based community detection using the Louvain method in the n-level hypergraph partitioning framework *KaHyPar* (**Ka**rlsruhe **Hy**pergraph **Par**titioning) and modified the default coarsening algorithm [1] to respect the community structure.[1] The code is written in C++ and compiled using g++-5.2 with flags `-O3 -mtune=native -march=native`. We refer to the original algorithm as KaHyPar and to the community-aware versions as *CA(·)*, where · is replaced with the appropriate edge weight function. All versions use the default configuration of KaHyPar.

Instances. We evaluate our algorithm on a large collection of 294 hypergraphs [1, 56], which contains instances from three benchmark sets: the ISPD98 VLSI Circuit Benchmark Suite [4], the University of Florida Sparse Matrix Collection [21], and the international SAT Competition 2014 [11]. Sparse Matrices are translated into hypergraphs using the row-net model [16], i.e. each row is treated as a net and each column as a vertex. For

[1] Our implementation is available from `https://github.com/SebastianSchlag/kahypar`.

Table 1 Summary of the hypergraph collection used in the experiments. Instances marked with a * are newly added and were not part of the collection used in [1, 56].

Application	VLSI		Sparse Matrix	SAT Solving		
Benchmark Set	ISPD98	DAC2012	UF-SPM		SAT14	
Representation	direct	direct	row-net	literal	primal	dual
Density Class	$d \approx 1$	$d \approx 1$	$d \ll 1, d \approx 1, d \gg 1$	$d \gg 1$	$d \gg 1$	$d \ll 1$
Community Str.	✓	✓	some instances [3]	✓	✓	✓
# Hypergraphs	18	10*	184	92	92*	92*
# in Subset	10	5*	60	30	30*	30*

SAT instances, each boolean *literal* is mapped to one vertex and each clause constitutes a net [53]. In order to incorporate more recent VLSI circuits, we add the instances of the DAC 2012 Routability-Driven Placement Contest [62] to the benchmark set. Furthermore, each SAT instance is also converted into *primal* and *dual* representation [48], which are more common in the SAT solving community than the literal model proposed in [53]. In the *primal* model each variable is represented by a vertex and each clause is represented by a net, whereas in the *dual* model the opposite is the case. While it is known that VLSI circuits and complex networks like web graphs and social networks have a naturally existing clustering structure [25, 27], recent work [7, 29] suggests the same for industrial SAT instances. The complete benchmark set consists of 488 hypergraphs with unit vertex and net weights.[2] It is used to compare our community-aware algorithm to KaHyPar and to other systems. To study the effects of edge weights on the solution quality for hypergraphs with different densities we use the representative subset of 100 hypergraphs proposed in [56] and add the five smallest DAC2012 hypergraphs as well as the primal and dual representation of each literal SAT hypergraph. In total, the subset therefore consists of 165 hypergraphs, which we divide in three density classes. The class $d \ll 1$ is comprised of all hypergraphs with $d < 0.75$. Hypergraphs with $0.75 \leq d \leq 1.25$ form class $d \approx 1$, while hypergraph with $d > 1.25$ are assigned to class $d \gg 1$. An overview of our benchmark sets is given in Table 1. While VLSI hypergraphs have $|V| \simeq |E|$ and therefore $d \simeq 1$ [18, 53], SAT hypergraphs exhibit different densities. A primal (or literal) hypergraph of a SAT formula with n variables and $m \in \mathcal{O}(n)$ clauses has density $d \gg 1$, while its dual representation has $d \ll 1$. Instances derived from sparse matrices cover all three cases.

All hypergraphs are partitioned into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks with $\varepsilon = 0.03$. For each value of k, a k-way partition is considered to be *one* test instance, resulting in a total of 1155 instances for experiments on the subset and 3416 instances for the full benchmark set.

System and Methodology. All experiments are performed on a single core of a machine consisting of two Intel Xeon E5-2670 Octa-Core processors (Sandy Bridge) clocked at 2.6 GHz. The machine has 64 GB main memory, 20 MB L3- and 8x256 KB L2-Cache and is running RHEL 7.2. To show the effect of community-aware coarsening on the performance of KaHyPar relative to state-of-the-art HGP tools, we compare it with the k-way (hMetis-K) and the recursive bisection variant (hMetis-R) of hMetis 2.0 (p1) [41, 42], and to PaToH 3.2 [16]. These HGP libraries were chosen because they provide the best solution quality [1]. hMetis

[2] The complete benchmark set along with detailed statistics for each hypergraph is publicly available from `http://algo2.iti.kit.edu/schlag/sea2017/`.
[3] Our benchmark set includes hypergraphs derived from web crawls and social networks.

does not directly optimize the $(\lambda-1)$ metric. Instead it optimizes the *sum-of-external-degrees* (SOED), which is closely related to the connectivity metric: $(\lambda-1)(\Pi) = \text{SOED}(\Pi) - \text{cut}(\Pi)$ for unweighted hypergraphs (i.e., each cut net contributes λ times its weight to the objective). We therefore set both hMetis versions to optimize SOED and calculate the $(\lambda-1)$-metric accordingly. This approach is also used by the authors of hMetis-K [42]. hMetis-R defines the maximum allowed imbalance of a partition differently [41]. An imbalance value of 5, for example, allows each block to weigh between $0.45 \cdot c(V)$ and $0.55 \cdot c(V)$ *at each bisection step*. We therefore translate our imbalance parameter ε to ε' as described in Eq. (4) such that it matches our balance constraint after $\log_2(k)$ bisections:

$$\varepsilon' := 100 \cdot \left(\left((1+\varepsilon) \frac{\lceil \frac{c(V)}{k} \rceil}{c(V)} \right)^{\frac{1}{\log_2(k)}} - 0.5 \right). \tag{4}$$

PaToH is configured to use a final imbalance ratio of ε to match our balance constraint. Since PaToH ignores the random seed if configured to use the quality preset, we report both the result of the quality preset (PaToH-Q) and the average over ten repetitions using the default configuration (PaToH-D). All partitioners have a time limit of *eight* hours per test instance. We perform ten repetitions with different seeds for each test instance and report the *arithmetic mean* of the computed cut and running time as well as the best cut found. When averaging over different instances, we use the *geometric mean* in order to give every instance a comparable influence on the final result. In order to compare different algorithms in terms of solution quality, we perform a more detailed analysis using the performance plots introduced in [56]: For each algorithm, these plots relate the smallest minimum cut of all algorithms to the corresponding cut produced by the algorithm on a per-instance basis. For each algorithm, these ratios are sorted in increasing order. The plots use a cube root scale for both axes to reduce right skewness [20] and show $1 - (\text{best}/\text{algorithm})$ on the y-axis to highlight the instances were each partitioner performs badly. A point close to one indicates that the partition produced by the corresponding algorithm was considerably worse than the partition produced by the best algorithm. A value of zero therefore indicates that the corresponding algorithm produced the best solution. Points above one correspond to infeasible solutions that violated the balance constraint. Thus an algorithm is considered to outperform another algorithm if its corresponding ratio values are below those of the other algorithm. In order to include instances with a cut of zero into the results, we set the corresponding cut values to *one* for ratio computations. Furthermore, we conduct Wilcoxon matched pairs signed rank tests [64] (using a 1% significance level) to determine whether or not the difference of KaHyPar-CA and the other algorithms is statistically significant. At a 1% significance level, a Z-score with $|Z| > 2.58$ is considered significant.

Evaluation of Edge Weights. Figure 3 summarizes the results of our experiments on the benchmark subset using different edge weights for the bipartite graph edges. For each density class a box plot shows the improvement of KaHyPar-CA($\cdot$) over KaHyPar for initial cuts (computed by the initial partitioning algorithm) and the final average and best cuts (after uncoarsening and local search). Using uniform edge weights for low density hypergraphs worsens the solution quality. However, although the initial cuts are significantly worse in this case, the best cuts are only 2% worse on average than those of KaHyPar. This shows the strength of the n level approach combined with strong local search heuristics. Weighting schemes that encode structural information about the hypergraph into the edge weights perform significantly better. Both CA(ω_e) and CA(ω_{de}) ensure that the community structure of the bipartite graph is not dominated by high-degree $\perp$-nodes (large nets) by incorporating

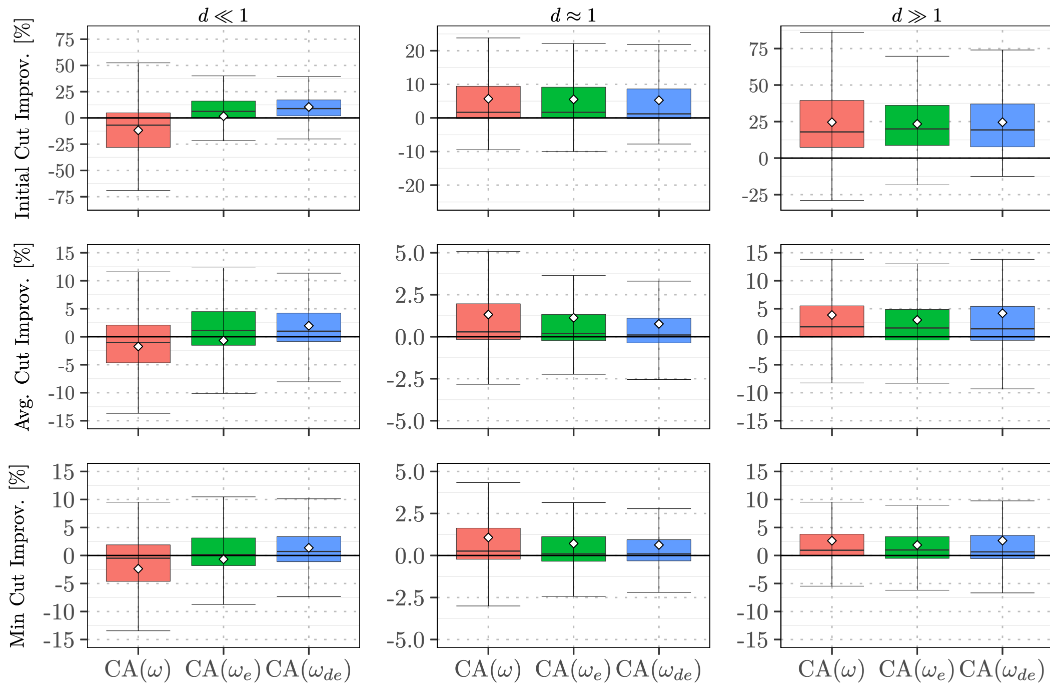


Figure 3 Comparing the improvement of KaHyPar-CA($\cdot$) (using different edge weighting schemes) over KaHyPar on the benchmark subset. Diamonds show the mean improvement.

Table 2 Improvement of KaHyPar-CA over KaHyPar on the benchmark subset. KaHyPar-CA uses $\omega(v, e)$ for hypergraphs with medium and high density and $\omega_{de}(v, e)$ for low-density hypergraphs.

Improvement [%]	VLSI		Sparse Matrix		SAT14		
	DAC2012	ISPD98	All	WebSocial	Primal	Literal	Dual
initial cut	20.5	13.8	4.1	24.8	23.8	34.0	12.2
min cut	3.9	2.0	0.8	3.5	3.5	4.0	1.6
average cut	4.7	2.3	1.1	5.5	4.8	5.7	2.2
worst cut	5.5	2.9	1.5	7.2	6.5	8.0	3.1

the net sizes into the edge weight. However, we can see that CA(ω_{de}) is more stable than CA(ω_e).

Its mean improvement is close to the median, always above zero, and always above the mean improvement of CA(ω_e), which shows that additionally strengthening the connection between $\perp$-nodes and high-degree $\top$-nodes indeed has a positive impact on solution quality. For hypergraphs with density $d \approx 1$ uniform edge weights perform best. If the density of the hypergraph is large, all three schemes give comparable results. This can be explained by the fact that if $d \gg 1$, most nets are small. This translates to "small stars" in the bipartite graph (or even paths for nets with $|e| = 2$), which do not distort the community structure of $\top$-nodes. Based on these results, we configure the final version of our algorithm to choose the weighting scheme at runtime depending on the observed density. If $d \geq 0.75$, it uses uniform edge weights, otherwise $\omega_{de}(v, e)$.[4] In the following we will refer to this configuration as KaHyPar-CA. As can be seen in Table 2 KaHyPar-CA significantly improves the initial

[4] Figure 5 in Appendix A compares all three edge weighting schemes and validates this decision.

Table 3 Comparing the average running times of KaHyPar-CA with KaHyPar and other tools.

Algorithm	Running Time [s]							
	All	DAC2012	ISPD98	Primal	Literal	Dual	SPM	WebSocial
KaHyPar	20.4	289.5	8.1	15.6	30.6	57.8	10.9	66.7
KaHyPar-CA	31.0	369.0	12.3	32.9	64.7	68.3	13.9	67.1
hMetis-R	79.2	446.4	29.0	66.2	142.1	200.4	41.8	89.7
hMetis-K	57.9	240.9	23.2	44.2	94.9	125.6	36.0	111.9
PaToH-Q	5.9	28.3	1.9	6.9	9.2	10.6	3.4	4.7
PaToH-D	1.2	6.5	0.4	1.1	1.6	2.9	0.8	0.9

cuts on all benchmark sets. The improvements in average cut (up to 5.7%) and min-cut (up to 4.0%) indicate that KaHyPar-CA is indeed able to compute better solutions than KaHyPar. Furthermore, the fact that the worst solutions of KaHyPar-CA are significantly better (up to 8.0%) than those of KaHyPar shows that community-aware coarsening improves the partitioner's robustness.

Comparison with other Systems. In the following, we exclude 194 out of 3416 instances because either PaToH-Q could not allocate enough memory or other partitioners did not finish in time. Excluded instances are shown in Appendix B. The following comparison is therefore based on the remaining 3222 instances.[5] As can be seen in Figure 4 and Table 4, KaHyPar-CA performs significantly better than KaHyPar on *all* benchmark sets. Looking at the solution quality of all systems across all instances (top left), KaHyPar-CA produced the best partitions for 1346 of the 3222 instances. It is followed by hMetis-R (882), KaHyPar (734) and hMetis-K (460). PaToH-D and PaToH-Q computed the best partitions for 163 instances. Note that for some instances multiple partitioners computed the same solution. Comparing the best solutions of KaHyPar-CA to each partitioner individually, KaHyPar-CA produced better partitions than PaToH-D, PaToH-Q, hMetis-K, KaHyPar, hMetis-R in 2849, 2833, 2084, 1979, 1937 cases, respectively.

By using community-aware coarsening, KaHyPar-CA performs best on each of the benchmark sets. As can be seen in Table 4, the difference in solution quality is statistically significant for all benchmark sets except DAC2012, where KaHyPar-CA is on par with hMetis-R. For hypergraphs derived from matrices of web graphs and social networks[6], KaHyPar-CA dominates all other systems by computing the best partitions for 86 of the 115 instances. Table 3 compares the running times of all partitioners. Although community detection using the Louvain method is itself a multilevel algorithm executed on the bipartite graph representation, KaHyPar-CA remains on average faster than hMetis.

5 Conclusions and Future Work

We describe an improved coarsening scheme for hypergraph partitioning that incorporates *global* information about the structure of the hypergraph by detecting communities in the

[5] **Interactive** visualizations of the performance plots and detailed per-instance results can be found on the website accompanying this publication: `http://algo2.iti.kit.edu/schlag/sea2017/`.

[6] Based on the following matrices: `webbase-1M`, `ca-CondMat`, `soc-sign-epinions`, `wb-edu`, `IMDB`, `as-22july06`, `as-caida`, `astro-ph`, `HEP-th`, `Oregon-1`, `Reuters911`, `PGPgiantcompo`, `NotreDame_www`, `NotreDame_actors`, `p2p-Gnutella25`, `Stanford`, `cnr-2000`.

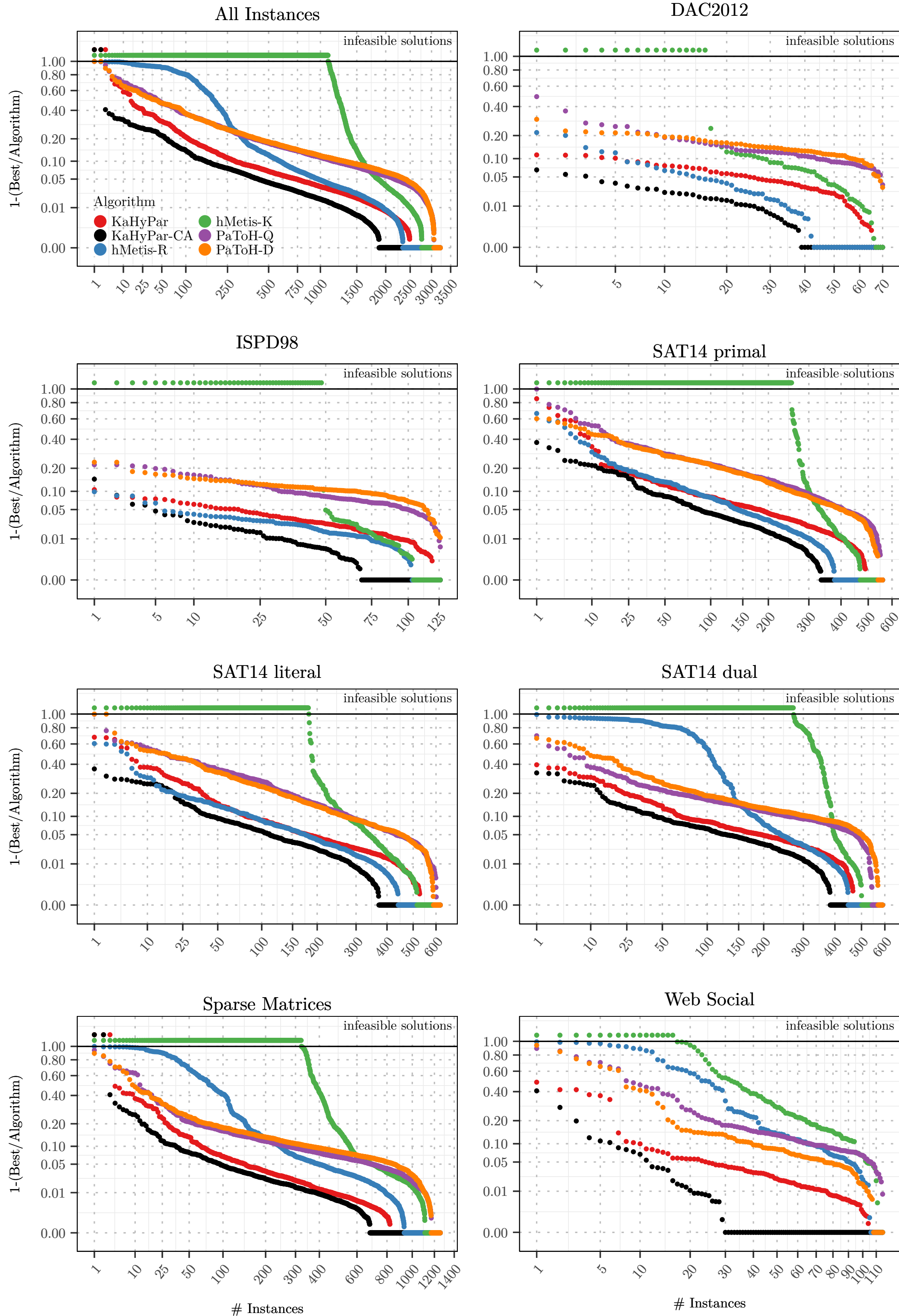


Figure 4 Min-Cut performance plots comparing KaHyPar-CA with KaHyPar and other systems. The y-axis shows the ratio between the smallest cut of all algorithms and the cut produced by the corresponding algorithm.

Table 4 Results of significance tests comparing KaHyPar-CA with KaHyPar and other systems on the full benchmark set. We report the Z-scores and p-values of the Wilcoxon matched pairs signed rank tests. At a 1% significance level, a Z-score with $|Z| > 2.58$ is considered significant. Negative Z-scores hereby indicate that KaHyPar-CA performs better than the respective algorithm. Note that hMetis-K has slight advantages in the following comparisons because we do not disqualify imbalanced partitions in the statistical analysis.

Class	Algorithm	KaHyPar-CA	
		Z	p
DAC2012	KaHyPar	−6.168	6.907e-10
	hMetis-R	−1.484	0.1379
	hMetis-K	−6.487	8.748e-11
	PaToH-D	−7.271	3.559e-13
	PaToH-Q	−7.271	3.559e-13
ISPD98	KaHyPar	−7.962	1.695e-15
	hMetis-R	−5.806	6.403e-09
	hMetis-K	−2.751	0.005935
	PaToH-D	−9.638	5.522e-22
	PaToH-Q	−9.636	5.655e-22
SAT14 Primal	KaHyPar	−11.22	3.232e-29
	hMetis-R	−4.411	1.027e-05
	hMetis-K	−6.918	4.579e-12
	PaToH-D	−17.23	1.56e-66
	PaToH-Q	−17.69	5.403e-70
SAT14 Literal	KaHyPar	−11.3	1.354e-29
	hMetis-R	−4.189	2.802e-05
	hMetis-K	−5.475	4.375e-08
	PaToH-D	−19.33	3.162e-83
	PaToH-Q	−19.56	3.12e-85
SAT14 Dual	KaHyPar	−7.271	3.573e-13
	hMetis-R	−8.339	7.515e-17
	hMetis-K	−8.071	6.969e-16
	PaToH-D	−18.21	4.656e-74
	PaToH-Q	−16.04	6.727e-58
UF-SPM	KaHyPar	−5.941	2.832e-09
	hMetis-R	−16.75	5.467e-63
	hMetis-K	−21.81	1.739e-105
	PaToH-D	−26.83	1.557e-158
	PaToH-Q	−25.39	3.612e-142
WebSocial	KaHyPar	−7.164	7.839e-13
	hMetis-R	−8.776	1.7e-18
	hMetis-K	−9.151	5.647e-20
	PaToH-D	−8.721	2.755e-18
	PaToH-Q	−7.368	1.737e-13

bipartite graph representation via modularity maximization using the Louvain method. We make this approach suitable for a wide spectrum of instances by appropriately choosing weights for the graph edges based on the density of the hypergraph. Experiments on a large benchmark set demonstrate that community-aware coarsening significantly improves the partitioning quality of KaHyPar on *all* instance classes, while having only a moderate impact on the overall running time. On all but one class, KaHyPar-CA performs statistically significantly better than KaHyPar, hMetis, and PaToH and is on par with the best partitioner otherwise.

There exist several ideas for future work. Given the significantly improved initial cuts, it might be feasible to equip KaHyPar with faster (and less strong) local search algorithms to narrow the gap between the running time of KaHyPar and PaToH. Modularity maximization is widely used to detect community structure but also known to exhibit a certain scaling behavior and resolution limit [26]. Future work therefore includes the analysis of whether these limitations negatively affect the coarsening process and if multi-resolution modularity [44] can be used as a remedy. Furthermore, there exist several alternative approaches to community detection such as Infomap [54] and Surprise [2] that could also be evaluated in our community-aware coarsening framework.

References

1. Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. Engineering a direct k-way hypergraph partitioning algorithm. In *19th Workshop on Algorithm Engineering and Experiments, (ALENEX)*, pages 28–42, 2017.
2. Rodrigo Aldecoa and Ignacio Marin. Deciphering Network Community Structure by Surprise. *PLOS ONE*, 6(9):1–8, 09 2011.
3. F. A. Aloul, I. L. Markov, and K. A. Sakallah. MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation. *Journal of Universal Computer Science*, 10(12):1562–1596, 2004.
4. C. J. Alpert. The ISPD98 Circuit Benchmark Suite. In *Proceedings of the 1998 International Symposium on Physical Design*, pages 80–85. ACM, 1998.
5. C. J. Alpert, J.-H. Huang, and A. B. Kahng. Multilevel Circuit Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, 1998.
6. C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: a Survey. *Integration, the VLSI Journal*, 19(1–2):1 – 81, 1995.
7. C. Ansótegui, J. Giráldez-Cru, and J. Levy. The community structure of sat formulas. In Alessandro Cimatti and Roberto Sebastiani, editors, *15th International Conference of Theory and Applications of Satisfiability Testing (SAT)*, pages 410–423. Springer, 2012.
8. C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level Direct K-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008.
9. D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Proc. Graph Partitioning and Graph Clustering – 10th DIMACS Implementation Challenge Workshop*, volume 588 of *Contemporary Mathematics*. AMS, 2013.
10. M. J. Barber. Modularity and community detection in bipartite networks. *Physical Review E*, 76:066102, Dec 2007.
11. A. Belov, D. Diepold, M. Heule, and M. Järvisalo. The SAT Competition 2014. `http://www.satcompetition.org/2014/`, 2014.
12. V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

13 U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. On Modularity Clustering. *IEEE Trans. Knowledge and Data Engineering*, 20(2):172–188, 2008.

14 T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992.

15 T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In *SIAM Conference on Parallel Processing for Scientific Computing*, pages 445–452, 1993.

16 Ü. V. Catalyürek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.

17 Ü. V. Catalyürek and C. Aykanat. PaToH: Partitioning Tool for Hypergraphs. `http://bmi.osu.edu/umit/PaToH/manual.pdf`, 1999.

18 J. Cong and S.K. Lim. Edge separability-based circuit clustering with application to multilevel circuit partitioning. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(3):346–357, 2004.

19 J. Cong and M. Smith. A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design. In *30th Conference on Design Automation*, pages 755–760, June 1993.

20 N. J. Cox. Stata tip 96: Cube roots. *Stata Journal*, 11(1):149–154(6), 2011. URL: `http://www.stata-journal.com/article.html?article=st0223`.

21 T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

22 K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Catalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Conference on Parallel and Distributed Processing*, IPDPS, pages 124–124. IEEE, 2006.

23 W. E. Donath. Logic partitioning. *Physical Design Automation of VLSI Systems*, pages 65–86, 1988.

24 V. Durairaj and P. Kalla. Guiding CNF-SAT Search via Efficient Constraint Partitioning. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, ICCAD, pages 498–501. IEEE, 2004.

25 S. Dutt and W. Deng. VLSI Circuit Partitioning by Cluster-removal Using Iterative Improvement Techniques. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD, pages 194–200. IEEE, 1996.

26 S. Fortunato and M. Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

27 S. Fortunato and D. Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.

28 Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

29 J. Giráldez-Cru and J. Levy. Generating sat instances with community structure. *Artificial Intelligence*, 238:119 – 134, 2016.

30 R. Guimerà, M. Sales-Pardo, and L. A. N. Amaral. Module identification in bipartite and directed networks. *Physical Review E*, 76:036102, Sep 2007.

31 S. W. Hadley. Approximation Techniques for Hypergraph Partitioning Problems. *Discrete Applied Mathematics*, 59(2):115–127, 1995.

32 L. Hagen and A. B. Kahng. A New Approach to Effective Circuit Clustering. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-aided Design*, ICCAD, pages 422–427. IEEE, 1992.

33 S. Hauck and G. Borriello. An Evaluation of Bipartitioning Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, Aug 1997.

34 B. Heintz and A. Chandra. Beyond graphs: Toward scalable hypergraph analysis systems. *ACM SIGMETRICS Performance Evaluation Review*, 41(4):94–97, April 2014.

35 B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *International Symposium on Solving Irregularly Structured Problems in Parallel*, pages 218–225, 1998.

36 B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

37 B. Hendrickson and R. Leland. A Multi-Level Algorithm For Partitioning Graphs. *SC Conference*, 0:28, 1995.

38 T. C. Hu and K. Moerder. Multiterminal Flows in a Hypergraph. In T.C. Hu and E.S. Kuh, editors, *VLSI Circuit Layout: Theory and Design*, chapter 3, pages 87–93. IEEE Press, 1985.

39 K. Suzuki and K. Wakita. Extracting Multi-facet Community Structure from Bipartite Networks. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE*, pages 312–319, 2009.

40 G. Karypis. Multilevel hypergraph partitioning. In Jason Cong and Joseph R. Shinnerl, editors, *Multilevel Optimization in VLSICAD*, pages 125–154. Springer, 2003.

41 G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration VLSI Systems*, 7(1):69–79, 1999.

42 G. Karypis and V. Kumar. Multilevel K-way Hypergraph Partitioning. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 343–348. ACM, 1999.

43 S. Klamt, U. Haus, and F. Theis. Hypergraphs and Cellular Networks. *PLOS Computational Biology*, 5(5):e1000385, 05 2009.

44 R. Lambiotte. Multi-scale modularity in complex networks. In *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, pages 546–553, May 2010.

45 A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80:056117, Nov 2009.

46 T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, Inc., 1990.

47 F. Lotfifar and M. Johnson. A Multi-level Hypergraph Partitioning Algorithm Using Rough Set Clustering. In *21st International Conference on Parallel and Distributed Computing (Euro-Par)*, pages 159–170, 2015.

48 Z. Mann and P. Papp. Formula partitioning revisited. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, pages 41–56. EasyChair, 2014.

49 T. Murata. Modularity for Bipartite Networks. In N. Memon, J. J. Xu, D. L. Hicks, and H. Chen, editors, *Data Mining for Social Network Data*. Springer, 2010.

50 N. Neubauer and K. Obermayer. Towards community detection in k-partite k-uniform hypergraphs. In *Proceedings of the NIPS 2009 Workshop on Analyzing Networks and Learning with Graphs*, pages 1–9, 2009.

51 M. E. J. Newman. Analysis of weighted networks. *Physical Review E*, 70:056131, Nov 2004.

52 M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, Feb 2004.

53 D. A. Papa and I. L. Markov. Hypergraph Partitioning and Clustering. In T. F. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics.* Chapman and Hall/CRC, 2007.

54 M. Rosvall, D. Axelsson, and C. T. Bergstrom. The map equation. *The European Physical Journal Special Topics*, 178(1):13–23, 2009.

55 S. E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, August 2007.

56 S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k-way Hypergraph Partitioning via n-Level Recursive Bisection. In *18th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 53–67, 2016.

57 D. G. Schweikert and B. W. Kernighan. A Proper Model for the Partitioning of Electrical Circuits. In *Proceedings of the 9th Design Automation Workshop*, DAC, pages 57–62. ACM, 1972.

58 A. Trifunovic. *Parallel Algorithms for Hypergraph Partitioning*. PhD thesis, University of London, 2006.

59 A. Trifunović and W. J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563 – 581, 2008.

60 Ü. V. Çatalyürek and M. Deveci and K. Kaya and B. Uçar. UMPa: A multi-objective, multi-level partitioner for communication minimization. In Bader et al. [9], pages 53–66.

61 B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005.

62 N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei. The DAC 2012 Routability-driven Placement Contest and Benchmark Suite. In *Proceedings of the 49th Annual Design Automation Conference*, DAC'12, pages 774–782. ACM, 2012.

63 S. Wichlund. On multilevel circuit partitioning. In *1998 International Conference on Computer-aided Design*, ICCAD, pages 505–511. ACM, 1998.

64 F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945. URL: `http://www.jstor.org/stable/3001968`.

A Performance Plots of Edge Weighting Schemes

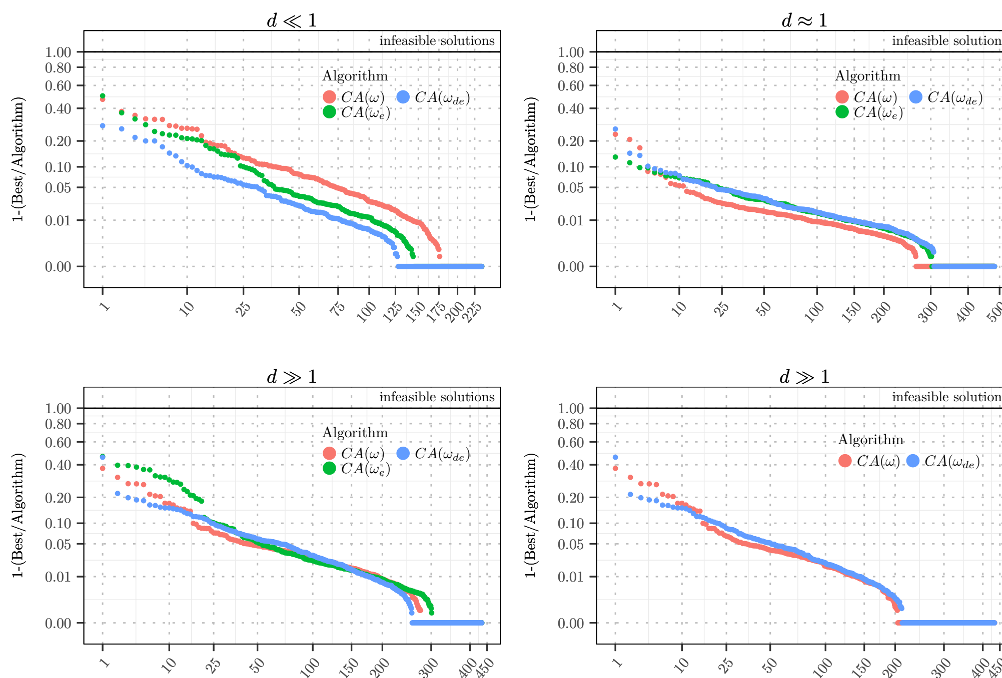


Figure 5 Min-Cut performance plots comparing the different edge weighting schemes of KaHyPar-CA on the benchmark subset.

B Excluded Test Instances

Out of 3416 test instances, we excluded the following 194 instances either because PaToH-Q could not allocate enough memory or one of the other partitioners could not partition the instances in the given time limit. The table is split into two groups: SAT instances and sparse matrix instances. Note that whenever KaHyPar or KaHyPar-CA exceeded the time limit, it was due to the long running time of local search.

Table 5 Instances excluded from the full benchmark set evaluation.

Hypergraph	2	4	8	16	32	64	128
10pipe-q0-k.dual				△	△	△	○△
10pipe-q0-k.primal	□	□	□	□	□	□	□
11pipe-k.dual	△	○△	○△	○△	○△	○△	○△
11pipe-k				○	○	○	○
11pipe-k.primal	□	□	□	□	□	□	○□
11pipe-q0-k.dual					△	○△	○△
11pipe-q0-k.primal	□	□	□	□	□	□	□
9dlx-vliw-at-b-iq3.dual							△
9dlx-vliw-at-b-iq3.primal	□	□	□	□	□	□	□
9vliw-m-9stages-iq3-C1-bug7.dual	△	●○△	●○△	●○△	●○△	●○△	●○△
9vliw-m-9stages-iq3-C1-bug7	△	△	●○△	●○△	●○△	●○□△	●○□△
9vliw-m-9stages-iq3-C1-bug7.primal	△	△		△	○△	○△	○△
9vliw-m-9stages-iq3-C1-bug8.dual	△	●○△	●○△	●○△	●○△	●○△	●○△
9vliw-m-9stages-iq3-C1-bug8	△	△	●○△	●○△	●○△	●○□△	●○□△
9vliw-m-9stages-iq3-C1-bug8.primal	△	△		△	○△	○△	○△
blocks-blocks-37-1.130-NOTKNOWN.dual		○	●○	●○	●○	●○	●○△
blocks-blocks-37-1.130-NOTKNOWN		□	□	□	□	□	□
blocks-blocks-37-1.130-NOTKNOWN.primal	□	□	□	□	□	□	□
E02F20.dual							○
E02F22.dual						○	○
openstacks-p30-3.085-SAT.primal	□	□	□	□	□	□	□
openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30-3.025-NOTKNOWN.primal	□	□	□	□	□	□	□
openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30-3.085-SAT.primal	□	□	□	□	□	□	□
transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.030-NOTKNOWN.dual							△

transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.050-NOTKNOWN.primal	□			□			□
q-query-3-L100-coli.sat.dual							△
q-query-3-L150-coli.sat.dual						△	△
q-query-3-L200-coli.sat.dual					△	△	△
q-query-3-L80-coli.sat.dual							△
velev-vliw-uns-2.0-uq5.dual			△	△	△	△	△
velev-vliw-uns-2.0-uq5.primal	□	□	□	□	□	□	□
velev-vliw-uns-4.0-9.dual					△	△	△
velev-vliw-uns-4.0-9.primal	□	□	□	□	□	□	□
192bit	□			□			
appu						❍	❍
ESOC	□	□			□	❍□	□
human-gene2					❍	❍	❍
IMDB				△	△	△	△
on-g500-logn16		△	△	△	△	❍△	❍△
Rucci1					□		
sls	□	□	□	❍□	❍□	❍□	❍□
Trec14							❍

△ :	KaHyPar/KaHyPar-CA exceeded time limit
● :	hMetis-R exceeded time limit
❍ :	hMetis-K exceeded time limit
□ :	PaToH-Q memory allocation error

Minimum Spanning Tree under Explorable Uncertainty in Theory and Experiments*

Jacob Focke[1], Nicole Megow[2], and Julie Meißner[3]

1 **Department of Computer Science, University of Oxford, Oxford, UK**
jacob.focke@cs.ox.ac.uk
2 **Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany**
nicole.megow@uni-bremen.de
3 **Institute of Mathematics, Technical University of Berlin, Berlin, Germany**
jmeiss@math.tu-berlin.de

Abstract

We consider the minimum spanning tree (MST) problem in an uncertainty model where uncertain edge weights can be explored at extra cost. The task is to find an MST by querying a minimum number of edges for their exact weight. This problem has received quite some attention from the algorithms theory community. In this paper, we conduct the first practical experiments for MST under uncertainty, theoretically compare three known algorithms, and compare theoretical with practical behavior of the algorithms. Among others, we observe that the average performance and the absolute number of queries are both far from the theoretical worst-case bounds. Furthermore, we investigate a known general preprocessing procedure and develop an implementation thereof that maximally reduces the data uncertainty. We also characterize a class of instances that is solved completely by our preprocessing. Our experiments are based on practical data from an application in telecommunications and uncertainty instances generated from the standard TSPLib graph library.

1998 ACM Subject Classification G.2.1 Combinatorial Algorithms, F.2.1 Computations on Discrete Structures

Keywords and phrases MST, explorable uncertainty, competitive ratio, experimental algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.22

1 Introduction

Uncertain data is a common issue in many real-world optimization problems. While it is clear that uncertain data can not be completely avoided, improved or exact data can often be explored at an additional cost. Classical approaches to optimization under uncertainty such as robust, stochastic, and online optimization do not capture this possibility. Uncertainty exploration takes a different approach by taking into account the exploration of uncertain data at extra cost. Here, the goal is to quantify the trade-off between an investment in more precise data and the resulting quality for the solution to the optimization problem. A major research line in this context asks for the minimum exploration cost to find an optimal solution. In a sense, this is the opposite of robust optimization that aims for the best solution with zero exploration cost.

* This research was carried out in the framework of Matheon supported by Einstein Foundation Berlin and the German Science Foundation (DFG) under contract ME 3825/1.

© Jacob Focke, Nicole Megow, and Julie Meißner;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 22; pp. 22:1–22:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Applications for optimization under explorable uncertainty can be found, e.g., in telecommunication or infrastructure network design, where either construction cost estimates can be improved through expert advice, or approximate connection lengths can be ensured through field measurements. Other optimization problems that allow uncertainty exploration are user demand estimates that can be improved through user surveys, and weather predictions that can be enhanced by both, additional measurements and more computational power.

In this paper, we consider the minimum spanning tree (MST) problem in the uncertainty exploration model. For each edge, we are given an uncertainty interval in which the exact edge weight lies. We can query each edge to find out its exact weight. The goal is to minimize the number of queries needed until we find a minimum spanning tree. As a performance measure we use the competitive ratio, that is, the worst-case ratio between the number of queries needed by an algorithm and the minimum number of queries required when given the exact data in advance. Precise definitions follow at the end of this section.

The MST problem, as one of the most fundamental and practically relevant combinatorial optimizations problems, has been investigated intensively in the uncertainty exploration model from the theoretical perspective. Several algorithms with provable worst-case guarantees are known [8, 15]. In this paper, we compare these algorithms theoretically, conduct the first practical experiments, and compare the theoretical and practical behavior of the algorithms. Furthermore, we investigate a preprocessing that was proposed in [15]. We develop an implementation thereof, for which we guarantee that it maximally reduces the data uncertainty. This is not only theoretically interesting but also practically relevant, as it reduces the data uncertainty that remains when starting an (arbitrary) algorithm.

We run our experiments on two different data sets. The first set of data is from a telecommunication service provider. It describes a problem that appears when expanding a cable network to a new roll-out area. First, the facility locations are chosen, that need to be connected. The exact connection costs between the facilities are unknown and can only be explored through costly field measurements. We find the best MST under uncertainty for these instances. We complement this practical data by a second data set, which we generate based on graphs available in the well-known graph library TSPLib [16].

Related Work. Optimization under uncertainty is an important, well-studied topic in theory and practice. The major lines of research are robust optimization [2], online optimization [4], and stochastic optimization [3], each modeling uncertain information in a different way. The first model where uncertain information can be explicitly explored at a fixed cost was studied by Kahan [14]. He investigated finding the maximum and median of a set of values known to lie in given uncertainty intervals. The recent survey by Erlebach and Hoffmann [6] gives a nice overview on research in the uncertainty exploration model. Various problems have been studied, including the k-th smallest value in a set of uncertainty intervals [14, 13, 10] and classical combinatorial optimization problems, such as shortest path [9], finding the median [10], the MST problem [8, 15, 11], the cheapest set problem [7] and the knapsack problem [12]. The latter work on the knapsack problem seems to be the only one that contains computational experiments conducted in this field.

The MST problem with uncertain edge weights was introduced by Erlebach et al. [8]. Their deterministic algorithm achieves an optimal competitive ratio of 2. A simplification of this algorithm that omits a repetitive restart by preserving the competitive ratio was given in [15]. Also the existence of a dual algorithm is observed. The main contribution in [15] is a randomized algorithm with expected competitive ratio of $1 + 1/\sqrt{2} \approx 1.707$ whereas the best-known lower bound is 1.5. The offline problem of finding the optimal query set for a given realization of edge weights can be solved in polynomial time [5].

Our Contribution. We theoretically compare the three algorithms for MST under uncertainty, make practical experiments and showcase similarities and differences between theoretical and practical observations. For the algorithms we define the best-possible preprocessing in Section 3 and characterize a class of instances which it solves completely. We observe exactly this structure in one of the practical data sets. Our experiments show that the average competitive ratio is small and the total number of queries as well. The comparison of theory and practice in Sections 2 and 5 shows that the competitive ratio and the variance in the size of an optimal solution are far from the worst-case. While theoretically there are instances on which the two deterministic algorithms show opposing behavior, in our experiments their performance is almost identical for all instances. We show that there are instances on which the two deterministic algorithms perform better than the randomized one. Surprisingly, we observe this behavior for the telecommunication data. For the TSPLib data the opposite happens and the randomized algorithm has a significantly smaller competitive ratio. Conducting the first experiments, we saw that the implementation hurdle is small and our run times are reasonably small, even though we did not optimize on it.

Problem Definition and Notation. Given an undirected, connected graph (V, E) with $|V| = n$ and $|E| = m$, we associate with each edge $e \in E$ an *uncertainty interval* A_e. This interval constitutes the only information about e's unknown weight $w_e \in A_e$. Such an interval is either *trivial*, $A_e = [L_e, U_e], L_e = U_e$, or it is *non-trivial*, i. e., $A_e = (L_e, U_e)$, with lower limit L_e and upper limit $U_e > L_e$. Closed, non-trivial intervals cannot be allowed as they lead to a non-constant competitive ratio [8]. We call an edge *trivial* if it has a trivial uncertainty interval, *non-trivial* otherwise. Let $\mathcal{A}$ be the set of uncertainty intervals for E. Then an instance of our problem is an *uncertainty graph* $G = (V, E, \mathcal{A})$ together with a *realization* $\mathcal{R}$ *of edge weights* $(w_e)_{e \in E}$ which lie in their corresponding uncertainty intervals, i. e., $w_e \in A_e$.

The task is to find a minimum spanning tree (MST) in the uncertainty graph G for an a priori unknown realization $\mathcal{R}$ of edge weights. We may query any edge $e \in E$ and obtain its exact weight w_e according to $\mathcal{R}$. The goal is to determine an MST through a sequence of queries that is as short as possible. The resulting set of queries $Q \subseteq E$ is *feasible*, if there is a spanning tree which is an MST for every realization of edge weights $w_e \in A_e$ for $e \in E \setminus Q$ and the weights w_e defined by $\mathcal{R}$ for $e \in Q$. We call this problem *MST under uncertainty*.

We evaluate our algorithms by standard competitive analysis. An algorithm is *c-competitive* if, for any realization $\mathcal{R} = (w_e)_{e \in E}$, the number of queries is at most c times the optimal query number. For a fixed realization $\mathcal{R}$, the optimal number of queries describes the minimum size of a query set that verifies an MST. The *competitive ratio* of an algorithm is the infimum over all c such that the algorithm is c-competitive. For randomized algorithms we compare the expected number of queries to the optimal number of queries.

2 Introduction of Algorithms and Theoretical Comparison

In this section we discuss known algorithms for MST under uncertainty. The first (deterministic) algorithm Ured was introduced by Erlebach et al. [8]. It achieves the best-possible competitive ratio of 2. Subsequently, two deterministic algorithms Cycle and Cut with competitive ratio 2 were given by Megow et al. [15]. Originally, they were presented for the more general problem of computing a minimum weight matroid basis in the uncertainty exploration model. Applying Cycle to computing an MST, it can be interpreted as a variant of Ured without repeated restarts and, thus, we consider here only the simplified variant. The randomized algorithm Random with competitive ratio 1.707 and a preprocessing were given also in [15]. Here the best known lower bound is 1.5.

In our paper we consider the three algorithms Cycle, Cut and Random, which we briefly describe below; pseudo-code can be found in the appendix. We apply the preprocessing on the input to *all* three algorithms. Details on the preprocessing follow in Section 3.

Deterministic algorithm Cycle. The algorithm Cycle is a worst-out greedy algorithm that is based on the following MST characterization: The largest-weight edge in a cycle is not in any MST. It starts out with a candidate minimum spanning tree and then iteratively considers the other edges. Each additional edge defines a cycle together with the candidate tree. On this cycle, the two edges with largest upper limit are queried repeatedly, until we either verify the additional edge has largest weight or we find an edge of larger weight on the cycle. In the latter case we improve the tree by exchanging the two edges.

Deterministic algorithm Cut. Cut is the dual algorithm to Cycle, that is defined by matroid duality. It uses that the minimum-weight edge in a cut is in an MST. Like in the previous algorithm, Cut starts with a candidate MST, but iteratively considers the tree edges. Deleting a tree edge defines a cut. On this cut we repeatedly query the two edges with smallest lower limit, until we either verify that the tree edge has the smallest weight in the cut or find an edge of smaller weight to replace the candidate tree edge.

Randomized algorithm. The randomized algorithm Random crucially needs a preprocessed instance with the following structural property: For any cycle appearing in the algorithm, any feasible query set contains either the edge with largest upper limit e or all edges with overlapping interval, i.e., whose uncertainty interval contains the lower limit L_e. The preprocessing, which we discuss in detail in Section 3, guarantees this property [15].

The algorithm Random uses the same structure as Cycle. Starting out with a candidate MST, it iteratively considers the remaining edges not in this tree. For each remaining edge, the algorithm inspects the (unique) cycle it closes in the MST to see which of its edges should be queried. The preprocessing yields that on each such cycle any feasible query set contains either the edge with the largest upper limit, say f, or all cycle edges whose intervals overlap with that of f. The algorithm either queries the largest edge or all overlapping edges at once. To balance this decision over several cycles closed during the algorithm, Random introduces a potential for each edge. In each cycle additional potential is distributed to all overlapping edges such that they reach an equal level. Depending on the resulting amount of potential, either these edges or the edge with largest upper limit are queried. This decision is randomized by comparing the potential to a randomly chosen uniform threshold.

2.1 Comparing the Deterministic Algorithms

We show that there are instances on which Cycle and Cut have an opposing performance, meaning that one algorithm is near-optimal and the other shows its worst-case performance. Intuitively, the instance is solved by querying the edges of a single cycle C and Cycle queries pairs of edges on C only. Cut, however, almost exclusively queries pairs with only one edge in C. The reverse holds for instances, in which it suffices to query the edges of a single cut.

Our graph class SP consists of a path of edges S and a set of parallel edges P, each of which closes a cycle with S. We give two realizations $\mathcal{R}_1$ and $\mathcal{R}_2$ in Figure 1.

For $\mathcal{R}_1$ the set S is the unique optimal query set and a query set is a feasible solution only if it contains S. The first cycle closed by the algorithm Cycle contains S and exactly one edge of P. It queries all edges on this cycle, which is a feasible solution of size $|S| + 1$. Cut on the other hand considers cuts of the form $P + \{s\}$ with a non-queried edge $s \in S$.

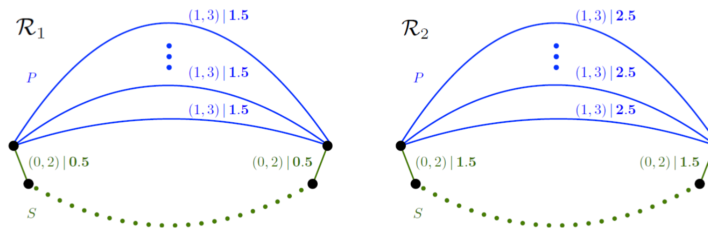


Figure 1 Different realizations for the class of uncertainty graphs SP lead to different extremes in the behavior of CYCLE and CUT. Edge labels: $(L_e, U_e)\,|\,\mathbf{w_e}$.

There are $|S|$ such cuts. For each, CUT queries a pair of edges as long as there are non-queried edges left in P. Thus, it queries $|S| + \min\{|S|, |P|\}$ edges. By choosing S and P of appropriate cardinality we can achieve every performance ratio $q \in (1, 2]$ for CUT. In particular, for $|S| \leq |P|$ and $|S| \to \infty$, the performance ratio of CYCLE approaches 1 and the ratio of CUT is 2.

The reverse holds for the realization $\mathcal{R}_2$. In this case a feasible query set has to contain P, CUT finds a solution of size $|P| + 1$, and CYCLE queries $|P| + \min\{|S|, |P|\}$ edges.

▶ **Observation 1.** *For any rational $q \in (1, 2]$, there exists a graph in the class SP and a realization such that* CYCLE *(*CUT*) is near-optimal whereas* CUT *(*CYCLE*) yields a performance ratio of q.*

Thus, theoretically the query set sizes can vary greatly for CYCLE and CUT. However, we do not observe this behavior for any of the instances in our experiments.

2.2 Comparing Randomized and Deterministic Algorithms

We show that RANDOM can be optimal for worst-case instances of CYCLE and CUT, and – somewhat surprisingly – the reverse is also possible.

Consider a family of cycles joined in one node, each with three edges f, g, h with uncertainty intervals $(1, 4)$, $(0, 3)$ and $[1, 1]$ respectively. Further, edge f has weight 3 and edge g has weight 1. Then, RANDOM terminates with a single query of either f or g in each cycle, while CYCLE and CUT query both f and g.

▶ **Observation 2.** *There are instances, for which* RANDOM *finds an optimal solution, while* CYCLE *and* CUT *achieve their worst-case ratio of* 2*.*

A similar instance evokes the reverse performance behavior. Consider a cycle C with k edges e_i with interval $(0, 3)$, one edge g with interval $(0, 4)$ and one edge f with interval $(1, 5)$. We choose the weights as $w_{e_i} = 2$ and $w_g = w_f = 3$. Then CYCLE and CUT query only edges f and g, which is optimal, but RANDOM yields its worst-case ratio $1 + 1/\sqrt{2}$ for $k \to \infty$.

▶ **Observation 3.** *There exists a family of uncertainty graphs, for which* CYCLE *and* CUT *perform optimally, whereas* RANDOM *asymptotically shows its worst case behavior.*

2.3 Variance of OPT

We investigate the variance of the optimal number of queries, OPT, under different realizations for a fixed input instance. We give an example instance in which small perturbations in the realization significantly change the value of OPT.

Algorithm 1 PREPROCESSING ($\prec_\ell, \prec_u$)

Input: An uncertainty graph $G = (V, E, \mathcal{A})$.
Output: A query set $Q \subseteq E$ and the two trees T_ℓ, T_u.
1: $Q \leftarrow \emptyset$.
2: Determine T_ℓ and T_u according to $\prec_\ell$ and $\prec_u$ respectively using Prim's algorithm [1].
3: **while** $T_\ell \setminus T_u$ contains a non-trivial edge **do**
4: Query all non-trivial edges in $T_\ell \setminus T_u$, and add them to Q.
5: Update T_ℓ and T_u.
6: **return** The query set Q and the two trees T_ℓ, T_u.

Consider a cycle C of length m consisting of an edge f with uncertainty interval $(1, 4)$ and $m-1$ identical edges $\{g_1, \dots, g_{m-1}\} =: \mathcal{G}$ with uncertainty interval $(0, 3)$ and weight 2. If we set the weight of f to be 3, it suffices to query f and OPT $= 1$. On the other hand, if the weight of f is 2, all edges in C have to be queried and OPT $= m$. Interestingly, we do not observe this large variance in our experiments (see Section 5: The optimal solution).

▶ **Observation 4.** *For a fixed uncertainty graph* OPT *can vary greatly even for minor changes of the underlying realization.*

3 Preprocessing

Preprocessing aims at simplifying the input instance, that is, we identify and query edges that must be queried by any algorithm including the optimal one. Naturally, we want to query as many such edges as possible before starting the actual algorithm.

There is a characterization of (a subset of) such edges that relies on the following definition. Given an instance of MST under uncertainty, the *lower limit tree* T_ℓ is an MST for the realization w^ℓ, in which all edge weights of edges with non-trivial uncertainty interval are arbitrarily close to their lower limits, more precisely $w^\ell_e = L_e + \varepsilon$ for infinitesimally small $\varepsilon > 0$. The *upper limit tree* $T_u \subseteq E$ is an MST for the realization in which edges have weights arbitrarily close to their upper limit, that is, $w^u_e = U_e - \varepsilon$. Note, that the order relation of edges with identical lower (upper) limit is yet unspecified. By $\prec_\ell$ and $\prec_u$ we denote an arbitrary but fixed pair of total orderings of edges, w.r.t. which we obtain a lower limit tree T_ℓ and upper limit tree T_u respectively.

▶ **Theorem 5** ([15]). *Given an uncertainty graph with lower and upper limit trees T_ℓ, T_u, any non-trivial edge $e \in T_\ell \setminus T_u$ is in every feasible query set for any realization.*

The preprocessing in [15] iteratively computes the trees T_ℓ and T_u and queries the edges in the set $T_\ell \setminus T_u$ until this set contains only edges with trivial uncertainty interval; see Alg. 1. The choice of $\prec_\ell$ and $\prec_u$ and thus specifying the order relation of edges with identical lower (upper) limit raises the potential for good or bad choices. As an example, consider a graph of k identical two-edge cycles that are all joined in one node. Each cycle is of the form $C = \{e_1, e_2\}$, where all edges have the same lower limit L and upper limits $U_1 < U_2$. Then, for any ordering $\prec_u$ the upper limit tree T_u does not contain e_2 for each of the cycles. For the lower limit ordering $\prec_\ell$, all orderings are feasible. For the ordering $e_1 \prec e_2$, we have $T_\ell = T_u$ and the preprocessing does not query any edge. However, for the ordering $e_2 \prec e_1$ the two trees are disjoint and k edges are queried in the first iteration of the preprocessing.

Observing this significant impact, we define a specific pair of total orderings $\prec_L, \prec_U$ on the edges and we prove that the algorithm above, PREPROCESSING ($\prec_L, \prec_U$), maximizes the total number of queries.

▶ **Definition 6** (Limit Orders and Trees). Let $G = (V, E, \mathcal{A})$ be an uncertainty graph and let $e_1, \dots, e_m$ be an arbitrary but fixed labeling of the edges in E. Then we define two orderings for the edges in E.

Lower Limit Order: $e_i \prec_L e_j$, if $L_{e_i} < L_{e_j}$ or if $L_{e_i} = L_{e_j}$ and one of the following holds:
- **(i)** e_i trivial and e_j non-trivial
- **(ii)** $U_{e_i} > U_{e_j}$ and e_j non-trivial
- **(iii)** $U_{e_i} = U_{e_j}$ and $i < j$.

Upper Limit Order: $e_i \prec_U e_j$, if $U_{e_i} < U_{e_j}$ or if $U_{e_i} = U_{e_j}$ and one of the following holds:
- **(i)** e_j trivial and e_i non-trivial
- **(ii)** $L_{e_i} > L_{e_j}$ and e_i non-trivial
- **(iii)** $L_{e_i} = L_{e_j}$ and $j < i$.

We call the corresponding lower and upper limit trees T_L and T_U.

We show that Preprocessing $(\prec_L, \prec_U)$ queries all edges which are in $T_\ell \setminus T_u$ for any other pair of orderings $\prec_\ell, \prec_u$. As a first step, it is not hard to see that an edge e, which is contained in $T_\ell \setminus T_u$ for some fixed orderings $\prec_\ell$ and $\prec_u$, remains in this set independently from queries of edges other than e.

▶ **Lemma 7.** *An edge in $T_\ell \setminus T_u$ remains in the set $T_\ell \setminus T_u$ until it is queried.*

Proof. Let e be in $T_\ell \setminus T_u$. As long as e is not queried, its interval limits do not change. Querying other edges only increases their lower limits and decreases their upper limits. Hence, e stays in T_ℓ and remains excluded from T_u. ◀

Next we show that Preprocessing $(\prec_L, \prec_U)$ does not terminate while there is a non-trivial edge in $T_\ell \setminus T_u$. The proof of Lemma 8 considers an edge e in $T_\ell \setminus T_u$ and proves the statement separately for the three cases $e \in T_L$, $e \notin T_L$ and $e \notin T_U$ as well as $e \in T_U \setminus T_L$.

▶ **Lemma 8.** *If there is a non-trivial edge in $T_\ell \setminus T_u$, then there is also one in $T_L \setminus T_U$.*

Proof. Assume there is a non-trivial edge $e \in T_\ell \setminus T_u$, but $T_L \setminus T_U$ contains only trivial edges. We distinguish three cases. If e is in T_L, it is also in T_U. Then there is an edge h, which is in the cut in $T_U \setminus e$ and in the cycle in $T_u \cup e$. As it is in the cut, we have $U_h \geq U_e$. At the same time, the cycle shows $U_h \leq U_e$, such that the two upper limits must be equal. Then, the fact that h is in the cut, but not in T_U means $L_e \geq L_h$. If h is trivial, e must also be trivial, which contradicts our assumption. Otherwise, as we choose $h \notin T_U$ and $T_L \setminus T_U$ contains only trivial edges, edge h is also not in T_L. If h is not in the cut $T_L \setminus e$, there must be an edge g in $T_L \setminus T_U$ that is in the cut $T_U \setminus e$ and in the cycle in $T_L \cup h$. This edge g is trivial, larger than e in the ordering $\prec_U$ and smaller than h in the ordering $\prec_L$. This means together with the observations about the bounds of e and h we made above, that we have $U_g \geq U_e = U_h$ and $L_e \geq L_h \geq L_g$. Thus e and h are both trivial: a contradiction. Alternatively we consider h is in the cut $T_L \setminus e$, where only edges at least as large as e are contained. This means $L_e \leq L_h$ and consequently $L_e = L_h$. The edge h is in the cut $T_L \setminus e$ and in the cut $T_U \setminus e$, which means we have $e \prec_L h$ and $e \prec_U h$. However, this contradicts that the intervals of e and h are identical.

If e is not in T_L and not in T_U, then there is an edge h, which is in the cut $T_\ell \setminus e$ and in the cycle in $T_L \cup e$. As it is in the cut, we have $L_e \leq L_h$ and h non-trivial, and as it is in the cycle we have $L_e \geq L_h$. Thus, we have $L_e = L_h$ and $U_h \geq U_e$ because of the ordering $\prec_L$. We choose $h \in T_L$. As $T_L \setminus T_U$ contains only trivial edges, edge h is also in T_U. If h is not in

the cycle $T_U \cup e$, there must be an edge g in $T_L \setminus T_U$ that is in the cut $T_U \setminus h$ and in the cycle $T_L \cup e$. This edge g is trivial, larger than h in the ordering $\prec_U$ and smaller than e in the ordering $\prec_L$. This means together with the observations about the bounds of e and h we made above, that we have $U_g \geq U_h \geq U_e$ and $L_h = L_e \geq L_g$. Thus e and h are both trivial: a contradiction. Alternatively we consider h is in the cycle $T_U \cup e$, where only edges with upper limit at most as large as e are contained. This means $U_h \leq U_e$ and consequently $U_h = U_e$. The edge h is in the cycle $T_U \cup e$ and in the cycle $T_L \cup e$, which means we have $h \prec_L e$ and $h \prec_U e$. However, this contradicts that the interval of e and h is identical.

Finally, we consider $e \in T_U \setminus T_L$. Then there is an edge h in the cut $T_U \setminus e$ and in the cycle $T_L \cup e$. This means $h \in T_L \setminus T_U$ and thus trivial. Additionally we have $e \prec_U h, h \prec_L e$, which means $L_h \leq L_e \leq U_e \leq U_h$. However, this is a contradiction as e is non-trivial. ◀

Combined, this means Preprocessing $(\prec_L, \prec_U)$ queries every non-trivial edge in $T_\ell \setminus T_u$.

▶ **Theorem 9.** Preprocessing *$(\prec_L, \prec_U)$ queries the union over all edges queried by* Preprocessing *$(\prec_\ell, \prec_u)$ for all orderings $\prec_\ell, \prec_u$. Thus, it queries the maximum number of edges characterized by Theorem 5.*

Proof. We show by induction over the number of iterations that edges queried in Preprocessing $(\prec_\ell, \prec_u)$ are also queried in Preprocessing $(\prec_L, \prec_U)$. By Lemma 7 and 8, all edges queried in iteration 1 of Preprocessing $(\prec_\ell, \prec_u)$ are also queried in our specific preprocessing. Let e be an edge queried in iteration $i > 1$ of Preprocessing $(\prec_\ell, \prec_u)$. Let S be the set of all edges queried in the previous iterations. Then, by induction, the set S is queried by Preprocessing $(\prec_L, \prec_U)$. Assume edge e is not queried by Preprocessing $(\prec_L, \prec_U)$. We consider Preprocessing $(\prec_\ell, \prec_u)$ in iteration i and additionally query all edges which are queried by Preprocessing $(\prec_L, \prec_U)$. By Lemma 7 edge e is still in $T_\ell \setminus T_u$ for this new uncertainty graph. However, this is exactly the uncertainty graph at the end of Preprocessing $(\prec_L, \prec_U)$. Thus, the termination of the algorithm at this point contradicts Lemma 8. ◀

3.1 Instances Solved by the Preprocessing

The preprocessing is a modification of the input instance and intuitively it simplifies it by removing uncertainty. We note, however, that in theory it can lead to a worse algorithm performance for specific input. Nevertheless, in our experiments, the preprocessing generally improves the performance ratio of our algorithms. One class of our data sets is even solved exactly by the preprocessing alone. We generalize this observation and characterize a family of uncertainty graphs which can be completely solved by our preprocessing.

▶ **Proposition 10.** *For uncertainty graphs, in which every cycle contains only edges with identical lower limit or only edges with identical upper limit,* Preprocessing $(\prec_L, \prec_U)$ *finds an optimal solution.*

Proof. The proof is by contradiction. Assume Preprocessing $(\prec_L, \prec_U)$ terminates with T_L and T_U and did not find a feasible query set. Then the uncertainty graph has a cycle C on which it is unclear which edge has the largest weight. All but one edge of C are in T_L. Assume, that originally all edges on the cycle had the same upper limit. If there is only one edge f with largest upper limit, all other edges on the cycle are trivial. Since it is unclear, which edge has largest weight on C, f cannot also have the largest lower limit. Thus f is non-trivial and in $T_L \setminus T_U$, which is a contradiction. Otherwise, there are two non-trivial edges e and f on C with largest upper limit, $e \in T_L$ and $f \notin T_U$. This means we can define

an alternative ordering $\prec_u$ with $f \prec_u e$ and thus $e \notin T_u$. Thus, for the preprocessing with orderings $\prec_L$ and $\prec_u$ we have $e \in T_L \setminus T_u$. By Theorem 9 this means PREPROCESSING $(\prec_L, \prec_U)$ queries e, a contradiction to e being non-trivial.

A cycle with identical lower limits can be treated analogously. ◀

4 Experimental Data

First, note that there is an inherent difficulty with practical experiments for exploration uncertainty. For a practical application the uncertainty intervals might be known as well as the exact edge weights of the *queried* edges. To decide the optimal number of queries necessary, in general one needs the exact edge weights of further edges. However, in practice there is no reason to explore additional edges after the solution has been found. Thus, even though we have practical data we need to generate a part of the instance.

Telecommunication. For the telecommunication data we have 5 different graphs of varying size with up to 1000 nodes available to us. For each of them we have two different sets of uncertainty intervals. In the first set, the terrain data, we consider the building cost uncertainty that arises from different terrains. The cost of a connection is limited by the construction cost per meter cable through a field and the cost under a paved street times the length of the connection. We draw the exact edge weight *uniformly* distributed in the interval. In this uncertainty setting, exploring the exact weight of an edge represents the time or cost investment it takes to identify the terrain of a particular connection. The second data set we call existence data. This setting assumes that the terrain of the connection is known, but it is uncertain if existing infrastructure is available or not. As a result the interval ranges from almost no building cost due to existing infrastructure to a fixed building cost, which is roughly known in advance. The exact edge weight follows a *two-point distribution* close to the two endpoints of the interval. We maintain the ratio of 20% small weight to 80% large weight that is observed in practice.

TSPLib. We consider the 19 graphs for the symmetric traveling salesman problem TSP of the library TSPLib that have at most 100 nodes. They are usually used for TSP computations, but we compute their minimum spanning trees. The library contains the exact edge weights and we need to create corresponding uncertainty intervals. We choose the interval size proportional to the weight of each edge, which is a natural approach that we also observe in the telecommunication data. We experiment with the ratio between interval size and exact edge weight, let us call this ratio d, to generate difficult instances. As before, we consider intervals such that the realization is either uniformly distributed or two-point distributed at the two extremes. For an edge with weight w we draw the lower limit L uniformly at random in $((1-d)\cdot w, w)$ in the uniform case and set the upper limit U to $L + d \cdot w$. In the extremal case we choose the lower limit close to the edge weight w such that $L < w$ or we choose the upper limit U close to w with $w < U$ each with probability $1/2$. Then we choose the other limit accordingly. We computed the average competitive ratio of all three algorithms for the two distributions and various values for d between 0.001 and 0.5 for 190 uncertainty graphs; see Figure 6. As we are interested in a worst-case behavior, we choose for our experiments a uniform value $d = 0.065$ for which all algorithms have a rather large competitive ratio.

As one aspect of our experimental analysis, we investigate for a given graph the variance of certain parameters. We distinguish between the two data types: For the telecommunication

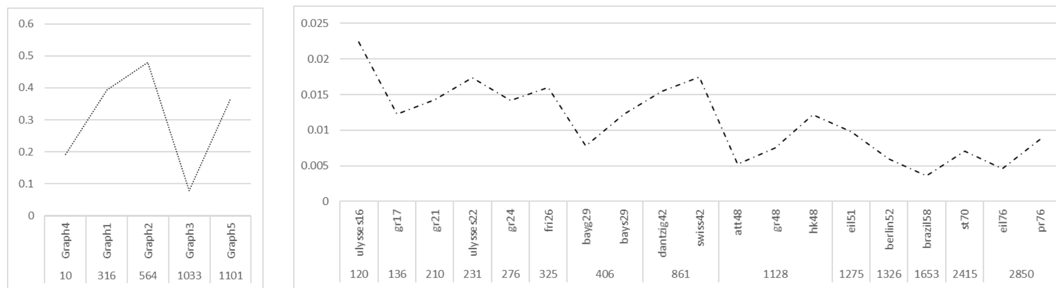


Figure 2 Size of the optimal solution OPT divided by the number of edges on the y-axis and the uncertainty graphs sorted by data set and increasing number of edges on the x-axis.

data the realization inside the uncertainty interval changes, while for the TSPLib data the location of the fixed length uncertainty interval around the also fixed realization changes.

5 Experimental Algorithm Analysis

For the detailed analysis we draw 100 uncertainty intervals/realizations for each graph in a data set, which yields 4800 instances in total. We perform our experiments with 20 repetitions of RANDOM per instance, as more repetitions did not alter the average performance. For each of the instances we compute the number of edges, the size of the query set in the preprocessing, the size of the optimal solution, the size of the query set for each of the three algorithms, the run time of the three algorithms as well as that of the preprocessing. For RANDOM we additionally compute the average number of edges on a cycle closed in the algorithm and the average number of edges on an algorithm cycle that have an uncertainty interval overlapping the one of the edge with largest upper limit. For the latter two parameters, we could not find a relation to the algorithm's performance. We summarize our experimental results in the following subsections. We make our code, the complete input and output data, and further analysis available at `http://www.coga.tu-berlin.de/fileadmin/i26/coga/MSTData.zip`.

5.1 The Optimal Solution

The size of the optimal solution OPT, that is, the minimum number of queries to find an MST, naturally grows with the size of the instance. To analyze a correlation, we consider the number of edges m as the instance size and determine the parameter OPT$/m$; see Figure 2. There are instances among the telecommunication data for which the ratio OPT$/m$ is as large as 0.5 and other ones where it is very small. Among this small number of instances the parameter behavior seems arbitrary. For the TSPLib data the ratio OPT$/m$ is a lot smaller and it decreases when m increases. Our theoretical analysis in Section 2.3 shows that for a single instance the behavior of this parameter can change between $1/m$ and 1. We do observe great variance for some instances of the telecommunication data, but very small variance for the TSPLib data. The variance is always far from the theoretical maximum variance.

5.2 Comparing Deterministic and Randomized Algorithms

For the telecommunication data the competitive ratio of all three algorithms has roughly the same average (cf. Figure 3). Averaging over all telecommunication instances CYCLE and CUT both have competitive ratio 1.18 and RANDOM has the slightly worse competitive ratio of

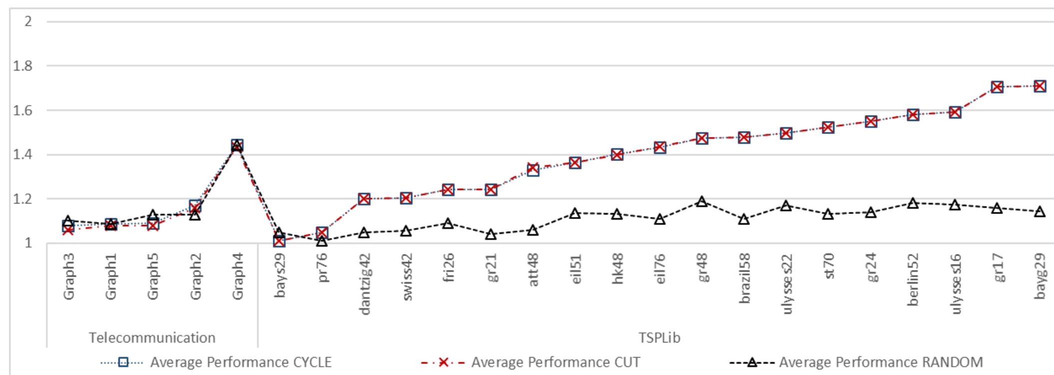


Figure 3 Average performance, i.e., the ratio of the algorithm query set size over the optimal query set size, for the three algorithms and the two data sets.

1.22. For the TSPLib data, the two deterministic algorithms have equal average competitive ratio 1.37, which is significantly larger than that for the telecommunication data. RANDOM has a notably smaller competitive ratio of 1.11 on average, that is even smaller than the ratio for the telecommunication data.

All average competitive ratios are far from their theoretical worst-case guarantee which is 2 for both deterministic algorithms and ≈ 1.707 for RANDOM. It is somewhat surprising, that despite the significant improvement of our randomized over the deterministic algorithms for the TSPLib data, there is no improvement for the telecommunication data. This means the usefulness of randomization depends on the considered data set. For the telecommunication data our way of randomization may even worsen the performance. This might seem counter-intuitive, but we give a theoretical explanation in Section 2.2.

5.3 Comparing the Deterministic Algorithms

In Section 2.1 we show that there can be a large difference between the performance ratio of CYCLE and CUT, even to the extreme case where one has ratio 1 and the other has ratio 2. However, as displayed in Figure 3, their average performance is identical for all graphs and both data sets. On an instance by instance comparison, the two ratios are equal for 98% of all instances we evaluate. The largest difference between performance ratios we observe in our experiments are seven instances with difference 0.33 and one with difference 0.7.

5.4 Variance in Performance

We compare the average performance of an algorithm to the worst performance among the best 25% of performances as well as the worst performance among the best 75% of all

show that the variance increases with the average performance ratio of an uncertainty graph and it is greater for the deterministic algorithms than for RANDOM. As the variance is equal for CYCLE and CUT, we only display the graph for CYCLE. For almost every graph individually, the variance between the different instances is very small.

5.5 Worst-Case Instances

Both deterministic algorithms have a competitive ratio of 2. In our experiments, this worst-case ratio is attained for some instances for which the optimal query number is at most 10. For the telecommunication data the worst-case is attained only on the pathological

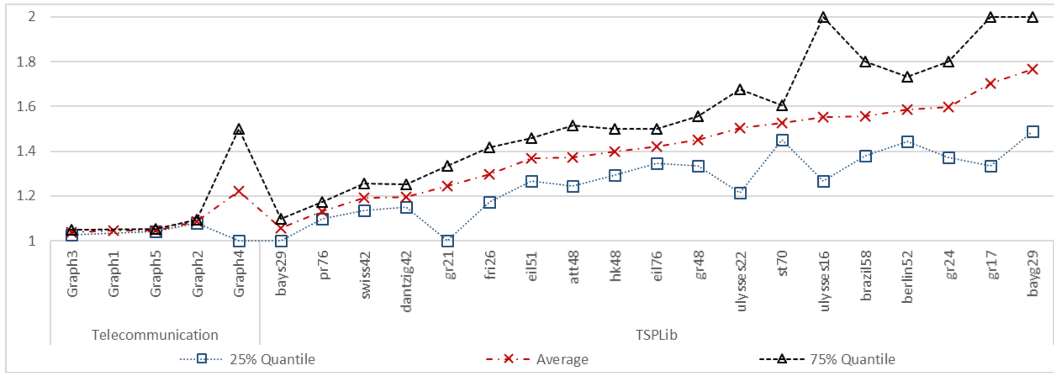


Figure 4 We show the variance of the average performance of CYCLE for each uncertainty graph by displaying the 25% Quantile, the average, and the 75% Quantile of the algorithm performance.

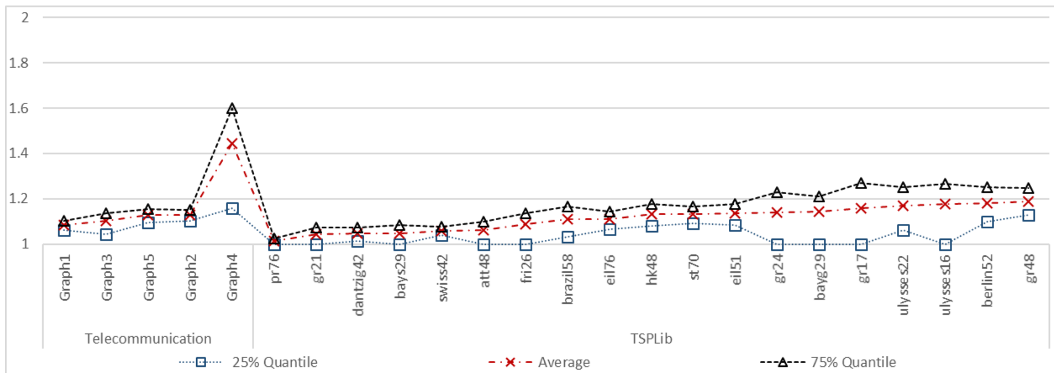


Figure 5 Variance of the average performance of RANDOM for each uncertainty graph with the 25% Quantile, the average, and the 75% Quantile of the algorithm performance.

example of Graph 4 consisting of a single cycle. However, the 7 smallest of the 19 graphs of the TSPLib data showcase instances with performance ratio 2. There are graphs, for which more than half of the instances showcase a worst-case ratio 2, but for others it is only a small percentage. The number of cases of ratio 2 roughly decreases with the number of edges in the graph. This is not symmetric to the case of performance ratio 1. There are more instances and more graphs for which there are instances which the algorithms solve optimally.

5.6 Comparing the Distributions

PREPROCESSING $(\prec_L, \prec_U)$ solves all telecommunication data instances with extreme distribution. We prove this theoretically in Section 3 and observe that this is due to the interval structure and not the distribution. In general, the share of instances solved by the preprocessing significantly increases from uniform to extreme distribution. For the TSPLib data the average share increases from 0.014 to 0.15 and for the telecommunication data it is 0.33 for the uniform distribution and 1 for the extreme one.

Additionally, we observe that the absolute number of queries almost always decreases, when changing from uniform to extreme distribution for all telecommunication instances and all algorithms. However, for the TSPLib data the behavior varies and for each graph there are instances where the uniform distribution has a smaller query number and others where the extreme distribution has a smaller query number.

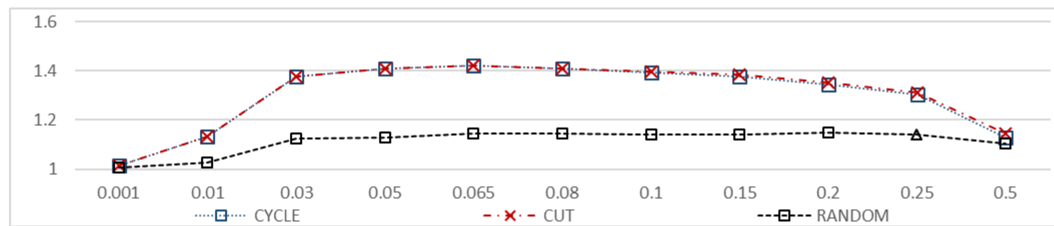


Figure 6 Average performance of the three algorithms for the TSPLib data for different values of the parameter $d =$ *interval size over exact edge weight*.

5.7 Interval Size

To create the TSPLib data, we experimented with different interval sizes. Figure 6 shows that the algorithms' performance changes greatly with the chosen ratio d of interval size over edge weight. For very large parameter d, almost all intervals overlap and their edges must be queried. For very small d, however, only few intervals overlap and almost no queries are required. This is true for any algorithm, and thus, it explains why Cycle, Cut and Random have an average competitive ratio near to 1 for very small and very large d.

5.8 Running Time

We run our experiments on a Linux system with an AMD Phenom II X6 1090T (3.2 GHz) processor and 8 GB RAM. Together, the three algorithms take about 1200 milliseconds to compute. The preprocessing dominates the run time with a duration of 770 milliseconds on average. On a one-by-one comparison Cycle and Random have similar average run times of around 20 milliseconds, but Cut's average run time is around 350 milliseconds. As expected, the run time increases with the graph size. In total, our data set of 400 instances up to a size of 70 vertices or 3000 edges can be generated and solved in roughly four hours. As we did not optimize the implementation in terms of run time, we expect that also larger instances can be solved in reasonable time.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.
2. A. Ben-Tal, L. El Ghaoui, and A. S. Nemirovski. *Robust Optimization*. Princeton Series in Applied Mathematics. Princeton University Press, 2009.
3. J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming*. Springer Series in Operations Research. Springer, 1997.
4. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
5. T. Erlebach and M. Hoffmann. Minimum spanning tree verification under uncertainty. In *Proceedings of WG*, pages 164–175, 2014. doi:10.1007/978-3-319-12340-0_14.
6. T. Erlebach and M. Hoffmann. Query-competitive algorithms for computing with uncertainty. *Bulletin of the EATCS*, Vol. 116, 2015.
7. T. Erlebach, M. Hoffmann, and F. Kammer. Query-competitive algorithms for cheapest set problems under uncertainty. *Th. Comp. Sci.*, 613:51–64, 2016. doi:10.1016/j.tcs.2015.11.025.

8 T. Erlebach, M. Hoffmann, D. Krizanc, M. Mihalák, and R. Raman. Computing minimum spanning trees with uncertainty. In *Proceedings of STACS*, pages 277–288, 2008. doi:10.4230/LIPIcs.STACS.2008.1358.

9 T. Feder, R. Motwani, L. O'Callaghan, C. Olston, and R. Panigrahy. Computing shortest paths with uncertainty. *Journal of Algorithms*, 62:1–18, 2007. doi:10.1016/j.jalgor.2004.07.005.

10 T. Feder, R. Motwani, R. Panigrahy, C. Olston, and J. Widom. Computing the median with uncertainty. *SIAM Journal on Computing*, 32:538–547, 2003. doi:10.1137/S0097539701395668.

11 J. Focke. Comparative analysis of algorithms for minimum spanning tree under uncertainty. Master's thesis, Technische Universität Berlin, 2017.

12 M. Goerigk, M. Gupta, J. Ide, A. Schöbel, and S. Sen. The robust knapsack problem with queries. *Computers & OR*, 55:12–22, 2015. doi:10.1016/j.cor.2014.09.010.

13 M. Gupta, Y. Sabharwal, and S. Sen. The update complexity of selection and related problems. *Theory Comput. Syst.*, 59(1):112–132, 2016. doi:10.1007/s00224-015-9664-y.

14 S. Kahan. A model for data in motion. In *Proceedings of STOC*, pages 267–277, 1991. doi:10.1145/103418.103449.

15 N. Megow, J. Meißner, and M. Skutella. Randomization helps computing a minimum spanning tree under uncertainty. *Journal on Scientific Computing*, 2017.

16 G. Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991. doi:10.1287/ijoc.3.4.376.

Faster Betweenness Centrality Updates in Evolving Networks*

Elisabetta Bergamini[1], Henning Meyerhenke[2], Mark Ortmann[3], and Arie Slobbe[4]

1 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
elisabetta.bergamini@kit.edu
2 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
meyerhenke@kit.edu
3 **University of Konstanz, Konstanz, Germany**
mark.ortmann@uni-konstanz.de
4 **Australian National University, Canberra, Australia**
arieslobbe1@gmail.com

Abstract

Finding central nodes is a fundamental problem in network analysis. Betweenness centrality is a well-known measure which quantifies the importance of a node based on the fraction of shortest paths going though it. Due to the dynamic nature of many today's networks, algorithms that quickly update centrality scores have become a necessity. For betweenness, several dynamic algorithms have been proposed over the years, targeting different update types (incremental- and decremental-only, fully-dynamic). In this paper we introduce a new dynamic algorithm for updating betweenness centrality after an edge insertion or an edge weight decrease. Our method is a combination of two independent contributions: a faster algorithm for updating pairwise distances as well as number of shortest paths, and a faster algorithm for updating dependencies. Whereas the worst-case running time of our algorithm is the same as recomputation, our techniques considerably reduce the number of operations performed by existing dynamic betweenness algorithms. Our experimental evaluation on a variety of real-world networks reveals that our approach is significantly faster than the current state-of-the-art dynamic algorithms, approximately by one order of magnitude on average.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases Graph algorithms, shortest paths, distances, dynamic algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.23

1 Introduction

Over the last years, increasing attention has been devoted to the analysis of complex networks. A common sub-problem for many graph based applications is to identify the most central nodes in a network. Examples include facility location [13], marketing strategies [12] and identification of key infrastructure nodes as well as disease propagation control and crime prevention [1]. As the meaning of "central" heavily depends on the context, various centrality measures have been proposed (see [4] for an overview). Betweenness centrality is a well-known measure which ranks nodes according to their participation in the shortest paths of the

* This work was partially supported by DFG grant ME-3619/3-1 (FINCA) and Br 2158/11-1 within the SPP 1736 *Algorithms for Big Data*. A. S. acknowledges support by the RISE program of DAAD.

© Elisabetta Bergamini, Henning Meyerhenke, Mark Ortmann, and Arie Slobbe;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 23; pp. 23:1–23:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

network. Formally, the betweenness of a node v is defined as $c_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths between two nodes s and t and $\sigma_{st}(v)$ is the number of these paths that go through node v. The fastest algorithm for computing betweenness centrality is due to Brandes [6], which we refer to as BA, from Brandes's algorithm. This algorithm is composed of two parts: an augmented APSP (all-pairs shortest paths) step, where pairwise distances and shortest paths are computed, and a dependency accumulation step, where the actual betweenness scores are computed. The augmented APSP is computed by running a SSSP (single-source shortest paths) computation from each node s and the dependency accumulation is performed by traversing only once the edges that lie in shortest paths between s and the other nodes. Therefore, BA requires $\Theta(|V||E|)$ time on unweighted and $\Theta(|V||E| + |V|^2 \log |V|)$ time on weighted graphs (i.e. the time of running n SSSPs).

Networks such as the Web graph and social networks continuously undergo changes. Since an update in the graph might affect only a small fraction of nodes, recomputing betweenness with BA after each update would be very inefficient. For this reason, several dynamic algorithms have been proposed over the last years [9, 14, 11]. As BA, these approaches usually solve two sub-tasks: the update of the augmented APSP data structures and the update of the betweenness scores. Although none of these algorithms is in general asymptotically faster than recomputation with BA, good speedups over BA have been reported for some of them, in particular for [11] and [14]. Nonetheless, an exhaustive comparison of these methods is missing in the literature.

In our paper, we only consider *incremental* updates, i.e. edge insertions or edge weight decreases (node insertions can be handled treating the new node as an isolated node and adding its neighboring edges one by one). Although it might seem reductive to only consider these kinds of updates, it is important to note that several real-world dynamic networks evolve only this way and do not shrink. For example, in a co-authorship network, a new author (node) or a new edge (coauthored publication) might be added to the network, but existing nodes or edges will not disappear. Another possible application is the centrality maximization problem, which consists in finding a set of edges that, if added to the graph, would maximize the centrality of a certain node. The problem can be approximated with a heuristic [7], which requires to add several edges to the graph and to recompute distances after each edge insertion.

Our contribution. We present a new algorithm for updating betweenness centrality after an edge insertion or an edge weight decrease. Our method is a combination of two contributions: a new dynamic algorithm for the augmented APSP, and a new approach for updating the betweenness scores. Based on properties of the newly-created shortest paths, our dynamic APSP algorithm efficiently identifies the node pairs affected by the edge update (i.e. those for which the distance and/or number of shortest paths change as a consequence of the update). The betweenness update method works by accumulating values in a fashion similar to that of BA. However, differently from BA, our method only processes nodes that lie in shortest paths between affected pairs.

We compare our new approach with two of the dynamic algorithms for which the best speedups over recomputation have been reported in the literature, i.e. KWCC [11] and KDB [14]. Compared to them, our algorithm for the augmented APSP update is asymptotically faster on dense graphs: $O(|V|^2)$ in the worst case versus $O(|V||E|)$. This is due to the fact that we iterate over the edges between affected nodes only once, whereas KDB and KWCC do it several times. Moreover, our dependency update works also for weighted graphs (whereas KDB does not) and it is asymptotically faster than the dependency update of KWCC for sparse graphs ($O(|V||E| + |V| \log |V|)$ in the worst case versus $O(|V|^3)$).

Our experimental evaluation on a variety of real-world networks reveals that our approach is significantly faster than both KDB and KWCC, on average by a factor 14.7 and 7.4, respectively.

2 Preliminaries

2.1 Notation

Let $G = (V, E, \omega)$ be a graph with node set $V = V(G)$, edge set $E = E(G)$ and edge weights $\omega : E \to \mathbb{R}_{>0}$. In the following we will use $n := |V|$ to denote the number of nodes and $m := |E|$ for the number of edges. Let $d(s,t)$ be the shortest-path distance between any two nodes $s, t \in V$. On a shortest path from s to t in G, we say w is a *predecessor* of t, or t is a *successor* of w, if $(w,t) \in E$ and $d(s,w) + \omega(w,t) = d(s,t)$. We denote the set of predecessors of t as $P_s(t)$. For a given source node $s \in V$, we call the graph composed of the nodes reachable from s and the edges that lie in at least one shortest path from s to any other node the *SSSP DAG of s*. We use σ_{st} to denote the number of shortest paths between s and t and we use $\sigma_{st}(v)$ for the number of shortest paths between s and t that go through v. Then, the betweenness centrality $c_B(v)$ of a node v is defined as: $c_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$.

Our goal is to keep track of the betweenness scores of all nodes after an update $(u, v, \omega'(u,v))$ in the graph, which could either be an edge insertion or an edge weight decrease. We use $G' = (V, E', \omega')$ to denote the new graph after the edge update and d', σ' and P' to denote the new distances, numbers of shortest paths and sets of predecessors, respectively. Also, we define the set of *affected sources* $S(t)$ of a node $t \in V$ as $\{s \in V : d(s,t) > d'(s,t) \vee \sigma_{st} \neq \sigma'_{st}\}$. Analogously, we define the set of affected targets of $s \in V$ as $T(s) := \{t \in V : d(s,t) > d'(s,t) \vee \sigma_{st} \neq \sigma'_{st}\}$. In the following we will assume G to be directed. However, the algorithms can be easily extended to undirected graphs.

2.2 Related Work

The basic idea of dynamic betweenness algorithms is to keep track of the old betweenness scores (and additional data structures) and efficiently update the information after some modification in the graph. Based on the type of updates they can handle, dynamic algorithms are classified as *incremental* (only edge insertions and weight decreases), *decremental* (only edge deletions and weight increases) or *fully-dynamic* (all kinds of edge updates). However, one commonality of all these approaches is that they build on the techniques used by BA [6], which we therefore describe in Section 3 in more detail.

The approach proposed by Green et al. [9] for unweighted graphs maintains all previously calculated betweenness values and additional information, such as pairwise distances, number of shortest paths and lists of predecessors of each node in the shortest paths from each source node $s \in V$. Using this information, the algorithm tries to limit the recomputation to the nodes whose betweenness has been affected by the edge insertion. Kourtellis et al. [14] modify the approach by Green et al. [9] in order to reduce the memory requirements from $O(nm)$ to $O(n^2)$. Instead of being stored, the predecessors are recomputed every time the algorithm requires them. The authors show that not only using less memory allows them to scale to larger graphs, but their approach (which we refer to as KDB, from the authors's initials) turns out to be also faster than the one by Green et al. [9] in practice (most likely because of the cost of maintaining the data structure of the algorithm by Green et al.).

Kas et al. [11] extend an existing algorithm for the dynamic all-pairs shortest paths (APSP) problem by Ramalingam and Reps [21] to also update betweenness scores. Differently

from the previous two approaches, this algorithm can handle also weighted graphs. Although good speedups have been reported for this approach, no experimental evaluation compares its performance with that of the approaches by Green et al. [9] and Kourtellis et al. [14]. We refer to this algorithm as KWCC, from the authors's initials.

Nasre et al. [19] compare the distances between each node pair before and after the update and then recompute the dependencies from scratch as in BA (see Section 3). Although this algorithm is faster than recomputation on some graph classes (i.e. when only edge insertions are allowed and the graph is sparse and weighted), it was shown in [3] that its practical performance is much worse than that of the algorithm proposed by Green et al. [9]. This is quite intuitive, since recomputing all dependencies requires $\Omega(n^2)$ time independently of the number of nodes that are actually affected by the insertion.

Pontecorvi and Ramachandran [20] extend existing fully-dynamic APSP algorithms with new data structures to update *all* shortest paths and then recompute dependencies as in BA. To our knowledge, this algorithm has never been implemented, probably because of the quite complicated data structures it requires. Also, since it recomputes dependencies from scratch as Nasre et al. [19], we expect its practical performance to be similar.

Differently from the other algorithms, the approach by Lee et al. [16] is not based on dynamic APSP algorithms. The idea is to decompose the graph into its biconnected components and then recompute the betweenness values from scratch only for the nodes in the component affected by the update. Although this allows for a smaller memory requirement ($\Theta(m)$ versus $\Omega(n^2)$ needed by the other approaches), the speedups on recomputation reported in [16] are significantly worse than those reported for example by Kourtellis et al. [14].

To summarize, KDB [14] and KWCC [11] are the most promising methods for a comparison with our new algorithm. For this reason, we will describe them in more detail in Section 4 and Section 5 and evaluate them in our experiments.

Since computing betweenness exactly can be too expensive for large networks, several approximation algorithms and heuristics have been introduced in the literature [5, 8, 22, 23] and, recently, also dynamic algorithms that update an approximation of betweenness centrality have been proposed [2, 3, 10, 23]. However, we will not consider them in our experimental evaluation since our focus here is on exact methods.

3 Brandes's algorithm (BA)

Betweenness centrality can be easily computed in time $\Theta(n^3)$ by simply applying its definition. In 2001, Brandes proposed an algorithm (BA) [6] which requires time $\Theta(nm)$ for unweighted and $\Theta(n(m+n\log n))$ for weighted graphs, i.e. the time of computing n single-source shortest paths (SSSPs). The algorithm is composed of two parts: the *augmented APSP* computation phase based on n SSSPs and the *dependency accumulation* phase. As dynamic algorithms based on BA build on these two steps as well, we explain them now in more detail.

Augmented APSP. In this first part, BA needs to perform an *augmented* APSP, meaning that instead of simply computing distances between all node pairs (s, t), it also finds the number of shortest paths σ_{st} and the set of predecessors $P_s(t)$. This can be done while computing an SSSP from each node s (i.e. BFS for unweighted and Dijkstra for weighted graphs). When a node w is extracted from the SSSP (priority) queue, BA computes $P_s(w)$ as $\{v : (v, w) \in E \wedge d(s, w) = d(s, v) + \omega(v, w)\}$ and σ_{sw} as $\sum_{v \in P_s(w)} \sigma_{sv}$.

Dependency accumulation. Brandes defines the one-side dependency of a node s on a node v as $\delta_{s\bullet}(v) := \sum_{t \neq v} \sigma_{st}(v)/\sigma_{st}$. It can be proven [6] that

$$\delta_{s\bullet}(v) = \sum_{w:v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_{s\bullet}(w)), \qquad \forall s, v \in V\,. \tag{1}$$

Intuitively, the term $\delta_{s\bullet}(w)$ in Eq. (1) represents the contribution of the sub-DAG (of the SSSP DAG of s) rooted in w to the betweenness of v, whereas the term 1 is the contribution of w itself. For all nodes v such that $\{w : v \in P_s(w)\} = \emptyset$ (i.e. the nodes that have no successors), we know that $\delta_{s\bullet}(v) = 0$. Starting from these nodes, we can compute $\delta_{s\bullet}(v)\ \forall v \in V$ by "walking up" the SSSP DAG rooted in s, using Eq. (1). Notice that it is fundamental that we process the nodes in order of decreasing distance from s, because to correctly compute $\delta_{s\bullet}(v)$, we need to know $\delta_{s\bullet}(w)$ for all successors of v. This can be done by inserting the nodes into a stack as soon as they are extracted from the SSSP (priority) queue in the first step. The betweenness of v is then simply computed as $\sum_{s \neq v} \delta_{s\bullet}(v)$.

4 Dynamic augmented APSP

As mentioned in Section 3, also dynamic algorithms based on BA build on its two steps. In the following, we will see how KDB [14] and KWCC [11] update the augmented APSP data structures (i.e. distances and number of shortest paths) after an edge insertion or a weight decrease. One difference between these two approaches is that KDB does not store the predecessors explicitly, whereas KWCC does. However, since in [14] it was shown that keeping track of the predecessors only introduces overhead, we report a slightly-modified version of KWCC that recomputes them "on the fly" when needed (we will also use this version in our experiments in Section 7). We will then introduce our new approach in Section 4.3.

4.1 Algorithm by Kourtellis et al. (KDB)

Let (u, v) be the new edge inserted into G (we recall that KDB works only on unweighted graphs, so edge weight modifications are not supported). For each source node $s \in V$, there are three possibilities: (i) $d(s, u) = d(s, v)$, (ii) $|d(s, u) - d(s, v)| = 1$ and (iii) $|d(s, u) - d(s, v)| > 1$ (in case (ii) and (iii), let us assume that $d(s, u) < d(s, v)$ without loss of generality). We recall that d is the distance *before* the edge insertion.

In the first case, it is easy to see that the insertion does not affect any shortest path rooted in s, and therefore nothing needs to be updated for s.

In case (ii), the distance between s and the other nodes is not affected, since there already existed an alternative shortest-path from s to v. However, the insertion creates new shortest paths from s to to v and consequently to all the nodes t in the sub-DAG (of the SSSP DAG from s) rooted in v. To account for this, for each of these nodes t, we add $\sigma_{su} \cdot \sigma_{vt}$ to the old value of σ_{st} (where $\sigma_{su} \cdot \sigma_{vt}$ is the number of new shortest paths between s and t going through (u, v)).

Finally, in case (iii), a part of the sub-DAG rooted in v might get closer to s. This case is handled with a BFS traversal rooted in v. In the traversal, all neighbors y of nodes x extracted from the BFS queue are examined and all the ones such that $d(s, y) \geq d'(s, x)$ are also enqueued. For each traversed node y, the new distance $d'(s, y)$ is computed as $\min_{z:(z,y) \in E} d'(s, z) + 1$ and the number of shortest paths σ'_{sy} as $\sum_{z \in P'_s(y)} \sigma_{sz}$.

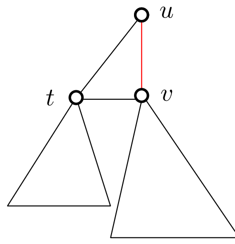


Figure 1 Insertion of (u, v).

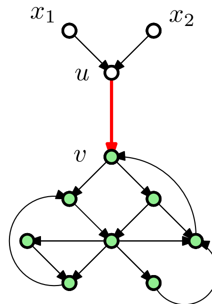


Figure 2 Affected targets (in green) and affected sources (x_1, x_2, u).

4.2 Algorithm by Kas et al. (KWCC)

KWCC updates the augmented APSP based on a dynamic APSP algorithm by Ramalingam and Reps [21]. Instead of checking for each source s whether the new edge (or the weight decrease) changes the SSSP DAG rooted in s, KWCC first identifies the *affected sources* $S = \{s : d(s, v) \geq d(s, u) + \omega'(u, v)\}$. These are exactly the nodes for which there is some change in the SSSP DAG. The affected sources are identified by running a pruned BFS rooted in u on G transposed (i.e. the graph obtained by reversing the direction of edges in G). For each node s traversed in the BFS, KWCC checks whether the neighbors of s are also affected sources and, if not, it does not continue the traversal from them. Notice that even on weighted graphs, a (pruned) BFS is sufficient since we already know all distances to v and we can basically sidestep the use of a priority queue.

Once all affected sources s are identified, KWCC starts a pruned BFS rooted in v for each of them. In the pruned BFS, only nodes t such that $d(s, t) \geq d(s, u) + \omega'(u, v) + d(v, t)$ are traversed (the *affected targets* of s). The new distance $d'(s, t)$ is set to $d(s, u)+\omega'(u, v)+d(v, t)$ and the new number of shortest paths $\sigma'(s, t)$ is set to $\sum_{z \in P'_s(t)} \sigma_{sz}$ as in KDB. Compared to KDB, the augmented APSP update of KWCC requires fewer operations. First, it efficiently identifies the affected sources instead of checking all nodes. Second, in case (iii), KDB might traverse more nodes than KWCC. For example, assume (u, v) is a new edge and the resulting SSSP DAG of u is as in Figure 1. Then, KWCC will prune the BFS in t, since $d(u, t) < d(u, v) + d(v, t)$, skipping all the SSSP DAGs rooted in t. On the contrary, KDB will traverse the whole subtree rooted in t, although neither the distances nor the number of shortest paths from u to those nodes are affected. The reason for this will be made clearer in Section 5.1.

4.3 Faster augmented APSP update

To explain our idea for improving the APSP update step, let us start with an example, shown in Figure 2. The insertion of (u, v) decreases the distance from nodes x_1, x_2, u to all the nodes shown in green. KWCC would first identify the affected sources $S = \{x_1, x_2, u\}$ and, for each of them, run a pruned BFS rooted in v. This means we are repeating almost exactly the same procedure for each of the affected sources. We clearly have to update the distances and number of shortest paths between each affected source and the affected targets (and this cannot be avoided). However, KWCC also goes through the outgoing edges of each affected target multiple times, leading to a worst-case running time of $O(mn)$.[1] Our basic idea is to avoid this redundancy and is based on the following proposition (a similar result was proven also in [18]).

▶ **Proposition 1.** *Let $t \in V$ and $y \in P_v(t)$ be given. Then, $S(t) \subseteq S(y)$.*

Proof. Let s be any node in $S(t)$, i.e. either $d'(s,t) = d(s,t)$ and $\sigma'_{st} \neq \sigma_{st}$ (case (i)), or $d'(s,t) < d(s,t)$ (case (ii)). We want to show that $s \in S(y)$.

Before proving this, we show that y has to be in $P'_s(t)$. In fact, if $s \in S(t)$, there have to be shortest paths between s and t going through (u, v), i.e. $d'(s,t) = d(s,u)+\omega'(u,v)+d(v,t)$. On the other hand, we know $y \in P_v(t)$ and thus

$$d'(s,t) = d(s,u) + \omega'(u,v) + d(v,y) + \omega(y,t). \tag{2}$$

Now, $d(s,u)+\omega'(u,v)+d(v,y)$ cannot be larger than $d'(s,y)$, or this would mean that $d'(s,t) > d'(s,v) + \omega(y,t)$, which contradicts the triangle inequality. Also, $d(s,u) + \omega'(u,v) + d(v,y)$ cannot be smaller than $d'(s,y)$ by definition of distance. Thus, $d'(s,y) = d(s,u) + \omega'(u,v) + d(v,y)$. If we substitute this in Eq. (2), we obtain $d'(s,t) = d'(s,y) + \omega(y,t)$, which means $y \in P'_s(t)$.

Now, let us consider case (i). We have two options: either y was a predecessor of t from s also before the edge update, i.e. $y \in P_s(t)$, or it was not. If it was not, it means $d(s,y) + \omega(y,t) > d(s,t) = d'(s,t) = d'(s,y) + \omega(y,t)$, which implies $d(s,y) > d'(s,y)$ and thus $s \in S(y)$. If it was, we can similarly show that $d(s,y) = d'(s,y)$. Since we have seen before that $d'(s,y) = d(s,u) + \omega'(u,v) + d(v,y)$, there has to be at least one new shortest path from s to y in G' going through (u,v), which means $\sigma'_{sy} > \sigma_{sy}$ and therefore $s \in S(y)$.

Case (ii) can be easily proven by contradiction. We know $d(s,t) \leq d(s,y) + \omega(y,t)$ (by the triangle inequality) and that $\omega'(y,t) = \omega(y,t)$. Thus, if it were true that $d(s,y) = d'(s,y)$ then

$$d(s,t) \leq d(s,y) + \omega(y,t) = d'(s,y) + \omega(y,t) = d'(s,t), \tag{3}$$

which contradicts our hypothesis that $d'(s,t) < d(s,t)$ (case (ii)). Thus, $d(s,y) \neq d'(s,y)$. Since pairwise distances in G' can only be equal to or shorter than pairwise distances in G, $d(s,y) \neq d'(s,y)$ implies $d(s,y) > d'(s,y)$ and thus $s \in S(y)$. ◀

In particular, this implies that $S(t) \subseteq S(v)$ for each $t \in T(u)$. Consequently, it is sufficient to compute $S(v)$ and $T(u)$ once via two pruned BFSs. Our approach is described in Algorithm 1. The pruned BFS to compute $S(v)$ is performed in Line 3. Then, a pruned

[1] Notice that this is true also for KDB, with the difference that KDB starts a BFS from each node instead of first identifying the affected sources and that it also visits additional nodes.

Algorithm 1: Augmented APSP update.

```
Input   : Graph G = (V, E), edge insertion/weight decrease (u, v, ω'(u, v)), d(s, t),
          σ_st, ∀(s, t) ∈ V^2
Output  : Updated d'(s, t), σ'_st, ∀(s, t) ∈ V^2
Assume  : Initially d'(s, t) = d(s, t) and σ'_st = σ_st  ∀(s, t) ∈ V^2
1  vis(v) ← false ∀v ∈ V;
2  if ω'(u, v) ≤ d(u, v) then
3      S(v) ← findAffectedSources(G, (u, v, ω'(u, v)));
4      d(u, v) ← ω'(u, v);
5      Q ← ∅;
6      p(v) ← v;
7      Q.push(v);
8      vis(v) ← true;
9      while Q.length() > 0 do
10         t = Q.front();
11         foreach s ∈ S(p(t)) do
12             if d(s, t) ≥ d(s, u) + ω'(u, v) + d(v, t) then
13                 if d(s, t) > d(s, u) + ω'(u, v) + d(v, t) then
14                     d'(s, t) ← d(s, u) + ω'(u, v) + d(v, t);
15                     σ'_st ← 0;
16                 end
17                 σ'_st ← σ'_st + σ_su · σ_vt;
18                 if t ≠ v then
19                     S(t).insert(s);
20                 end
21             end
22         end
23         foreach w s.t. (t, w) ∈ E do
24             if not vis(w) and d(u, w) ≥ ω'(u, v) + d(v, w) then
25                 Q.push(w);
26                 vis(w) ← true;
27                 p(w) ← t;
28             end
29         end
30     end
31 end
```

BFS from v is executed, whereby for each $t \in T(u)$ we store one of its predecessors $p(t)$ in the BFS (Line 27).

Let $d^\star(s,t)$ be the length of a shortest path between s and t going through (u,v), i.e. $d^\star(s,t) := d(s,u)+\omega'(u,v)+d(v,t)$. To finally compute $S(t)$ all that is left to do is to test whether $d^\star(s,t) \leq d(s,t)$ for each $s \in S(p(t))$ once we remove t from the queue (Lines 11–22). Note that this implies that $S(p(t))$ was already computed. In case $d^\star(s,t) < d(s,t)$, the path from s to t via edge (u,v) is shorter than before and therefore we set $d'(s,t)$ to $d^\star(s,t)$ and σ'_{st} to $\sigma_{su} \cdot \sigma_{vt}$, since all new shortest paths now go through (u,v)). Also in case of equality $(d^\star(s,t) = d(s,t))$, s is in $S(t)$, since its number of shortest paths has changed. Consequently we set σ'_{st} to $\sigma_{st} + \sigma_{su} \cdot \sigma_{vt}$ (since in this case also old shortest paths are still valid). If $d^\star(s,t) > d(s,t)$, the edge (u,v) does not lie on any shortest path from s to t, hence $s \notin S(t)$ (and s is not added to $S(t)$ in Lines 18–20).

5 Dynamic dependency accumulation

After updating distances and number of shortest paths, dynamic algorithms need to update the betweenness scores. This means increasing the score of all nodes that lie in new shortest paths, but also decreasing that of nodes that used to be in old shortest paths between affected

nodes. Again, we will first see how KDB and KWCC update the dependencies and then we will present our new approach in Section 5.3.

5.1 Algorithm by Kourtellis et al. (KDB)

In addition to d and σ, KDB keeps track of the old dependencies $\delta_{s\bullet}(v)$ $\forall s, v \in V$. The dependency update is done in a way similar to BA (see Section 3). Also in this case, nodes v are processed in decreasing order of their new distance $d'(s, v)$ from s (otherwise it would not be possible to apply Eq. (1)). However, in this case we would only like to process nodes for which the dependency has actually changed. To do this, while still making sure that the nodes are processed in the right order, KDB replaces the stack used in BA with a bucket list. Every node that is traversed during the APSP update is inserted into the bucket list in a position equal to its new distance from s. Then, nodes are extracted from the bucket list starting from the ones with maximum distance. Every time a node v is extracted, we compute its new dependency as $\delta'_{s\bullet}(v) = \sum_{w:v\in P'_s(w)} \frac{\sigma'_{sv}}{\sigma'_{sw}}(1 + \delta'_{s\bullet}(w))$. Since we are processing the nodes in order of decreasing new distance, we can be sure that $\delta'_{s\bullet}(v)$ is computed correctly. The score of v is then updated by adding the new dependency $\delta'_{s\bullet}(v)$ and subtracting the old $\delta_{s\bullet}(v)$, which was previously stored. Also, all neighbors $y \in P'_s(v)$ that are not in the bucket list yet are inserted at level $d'(s, y) = d'(s, v) - 1$. Notice that, in the example in Figure 1, all the nodes in the sub-DAG of t are necessary to compute the new dependency of t, although they have not been affected by the insertion. This is why they are traversed during the APSP update.

5.2 Algorithm by Kas et al. (KWCC)

KWCC does not store dependencies. On the contrary, for every node pair (s, t) for which either $d(s, t)$ or σ_{st} has been affected by the insertion, all the nodes in the new shortest paths and the ones in the old shortest paths between s and t are processed. More specifically, starting from t, all the nodes $y \in P'_s(t)$ are inserted into a queue. When a node y is extracted, we increase its betweenness by $\sigma'(s, y) \cdot \sigma'(y, t)/\sigma'(s, t)$ (i.e. the fraction of shortest paths between s and t going through y). Then, also y enqueues all nodes in $P'_s(y)$ and the process is repeated until we reach s. Decreasing the betweenness of nodes in the old paths is done in a similar fashion, with the only difference that nodes in $P_s(y)$ are enqueued (instead of nodes in $P'_s(y)$) and that $\sigma(s, y) \cdot \sigma(y, t)/\sigma(s, t)$ is subtracted from the scores of processed nodes. Notice that the worst-case complexity of this approach is $O(n^3)$, whereas that of KDB is $O(nm)$. This cubic running time is due to the fact that, for each affected node pair (s, t) (at most $\Theta(n^2)$), there could be up to $\Theta(n)$ nodes lying in either one of the old or new shortest paths between s and t. (In the running time analysis of [14], this is represented by the term $|\sigma_{old}|I$.) This means that, if many nodes are affected, KWCC can even be slower than recomputation with BA. On the other hand, we have seen in Section 4.2 that KDB also processes nodes for which the betweenness has not changed (see Figure 1 and its explaination), which in some cases might result in a higher running time than KWCC.

5.3 Faster betweenness update

We propose a new approach for updating the betweenness scores. As KWCC, we do not store the old dependencies (resulting in a lower memory requirement) and we only process the nodes whose betweenness has actually been affected. However, we do this by accumulating contributions of nodes only once for each affected source, in a fashion similar to KDB. For an

affected source $s \in S$ and for any node $v \in V$, let us define $\Delta_{s,\bullet}(v)$ as $\sum_{t \in T(s)} \sigma_{st}(v)/\sigma_{st}$. This is the contribution of nodes whose old shortest paths from s went through v, but which have been affected by the edge insertion. Analogously, we can define $\Delta'_{s,\bullet}(v)$ as $\sum_{t \in T(s)} \sigma'_{st}(v)/\sigma'_{st}$. Then, the new dependency $\delta'_{s,\bullet}(v)$ can be expressed as:

$$\delta'_{s,\bullet}(v) = \delta_{s,\bullet}(v) - \Delta_{s,\bullet}(v) + \Delta'_{s,\bullet}(v) \,. \tag{4}$$

Notice that for all nodes $t \notin T(s)$, $\sigma'_{st} = \sigma_{st}$ and $\sigma'_{st}(v) = \sigma_{st}(v)$, therefore their contribution to $\delta_{s,\bullet}(v)$ is not affected by the edge update. The new betweenness $c'_B(v)$ can then be computed as $c_B(v) - \sum_{s \in S} \Delta_{s,\bullet}(v) + \sum_{s \in S} \Delta'_{s,\bullet}(v)$. The following theorem allows us to compute $\Delta_{s,\bullet}(v)$ and $\Delta'_{s,\bullet}(v)$ efficiently.

▶ **Theorem 2.** *For any $s \in T, v \in V$:*

$$\Delta_{s,\bullet}(v) = \sum_{w: v \in P_s(w) \wedge w \in T(s)} \sigma_{sv}/\sigma_{sw}(1 + \Delta_{s,\bullet}(w)) + \sum_{w: v \in P_s(w) \wedge w \notin T(s)} \sigma_{sv}/\sigma_{sw} \cdot \Delta_{s,\bullet}(w) \ .$$

Similarly:

$$\Delta'_{s,\bullet}(v) = \sum_{w: v \in P'_s(w) \wedge w \in T(s)} \sigma'_{sv}/\sigma'_{sw}(1 + \Delta'_{s,\bullet}(w)) + \sum_{w: v \in P'_s(w) \wedge w \notin T(s)} \sigma'_{sv}/\sigma'_{sw} \cdot \Delta'_{s,\bullet}(w) \ .$$

Proof. We prove only the equation for $\Delta_{s,\bullet}(v)$, the one for $\Delta'_{s,\bullet}(v)$ can be proven analogously. Let t be any node in $T(s)$, $t \neq v$. Then, $\sigma_{st}(v)/\sigma_{st}$ can be rewritten as $\sum_{w: v \in P_s(w)} \sigma_{st}(v,w)/\sigma_{st}$, where $\sigma_{st}(v,w)$ is the number of shortest paths between s and t going through both v and w. Then:

$$\Delta_{s,\bullet}(v) = \sum_{t \in T(s)} \sigma_{st}(v)/\sigma_{st} = \sum_{t \in T(s)} \sum_{w: v \in P_s(w)} \sigma_{st}(v,w)/\sigma_{st} = \sum_{w: v \in P_s(w)} \sum_{t \in T(s)} \sigma_{st}(v,w)/\sigma_{st} \ .$$

Now, of the σ_{sw} paths from s to w, there are σ_{sv} many that also go through v. Therefore, for $t \neq w$, there are $\frac{\sigma_{sv}}{\sigma_{sw}} \cdot \sigma_{st}(w)$ shortest paths from s to t containing both v and w, i.e. $\sigma_{st}(v,w) = \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \sigma_{st}(w)$. On the other hand, if $t = w$, $\sigma_{st}(v,w)$ is simply σ_{sv}. Therefore, we can rewrite the equation above as:

$$\begin{aligned}
&\sum_{w: v \in P_s(w) \wedge w \in T(s)} \left\{ \frac{\sigma_{sv}}{\sigma_{sw}} + \sum_{t \in T(s) - \{w\}} \frac{\sigma_{st}(v,w)}{\sigma_{st}} \right\} + \sum_{w: v \in P_s(w) \wedge w \notin T(s)} \sum_{t \in T(s)} \frac{\sigma_{st}(v,w)}{\sigma_{st}} \\
&= \sum_{w: v \in P_s(w) \wedge w \in T(s)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(1 + \sum_{t \in T(s) - \{w\}} \frac{\sigma_{st}(w)}{\sigma_{st}} \right) + \sum_{w: v \in P_s(w) \wedge w \notin T(s)} \frac{\sigma_{sv}}{\sigma_{sw}} \sum_{t \in T(s)} \frac{\sigma_{st}(w)}{\sigma_{st}} \\
&= \sum_{w: v \in P_s(w) \wedge w \in T(s)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \Delta_{s,\bullet}(w)) + \sum_{w: v \in P_s(w) \wedge w \notin T(s)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \Delta_{s,\bullet}(w) \ .
\end{aligned}$$ ◀

Theorem 2 allows us to accumulate the dependency changes in a way similar to BA. To compute $\Delta_{s,\bullet}$, we need to process nodes in decreasing order of $d(s,\cdot)$, whereas to compute $\Delta'_{s,\bullet}$ we need to process them in decreasing order of $d'(s,\cdot)$. To do this, we use two priority queues PQ_s and PQ'_s (if the graph is unweighted, we can use bucket lists as the ones used in KDB). Notice that nodes w such that $\sigma_{st}(w) = 0 \wedge \sigma'_{st}(w) = 0 \ \forall t \in T(s)$ do not need to be added to the queue. PQ_s and PQ'_s are filled with all nodes in $T(s)$ during the APSP update in Algorithm 1. In PQ_s, nodes w are inserted with priority $d(s,w)$ and PQ'_s with priority $d'(s,w)$. Algorithm 2 shows how we decrease betweenness of nodes that lied in old shortest

Algorithm 2: Betweenness update for nodes in old shortest paths.

```
1  Δ_{s,•}(u) ← 0 ∀u ∈ V;
2  while PQ_s ≠ ∅ do
3      w ← PQ_s.extractMax();
4      c_B(w) ← c_B(w) − Δ_{s,•}(w);
5      foreach y s.t. (y, w) ∈ E do
6          if y ≠ s and d(s, w) = d(s, y) + ω(y, w) then
7              if w ∈ T(s) then
8                  c ← (σ_sy / σ_sw) · (1 + Δ_{s,•}(w));
9              end
10             else
11                 c ← (σ_sy / σ_sw) · Δ_{s,•}(w);
12             end
13             if y ∉ PQ_s then
14                 Insert y into PQ_s with priority d(s, y);
15             end
16             Δ_{s,•}(y) ← Δ_{s,•}(y) + c;
17         end
18     end
19 end
```

paths from s (notice that this is repeated for each $s \in S(v)$). In Lines 7–12, Theorem 2 is applied to compute $\Delta_{s,\bullet}(y)$ for each predecessor y of w. Then, y is also enqueued and this is repeated until PQ_s is empty (i.e. when we reach s). The betweenness update of nodes in the new shortest paths works in a very similar way. The only difference is that PQ'_s is used instead of PQ, that d' and σ' are used instead of d and σ and that $\Delta'_{s,\bullet}$ is added to c_B and not subtracted in Line 4. At the end of the update, σ is set to σ' and d is set to d'.

In undirected graphs, we can notice that $\sum_{s \in S(w)} \Delta_{s,\bullet}(w) = \sum_{t \in T(w)} \Delta_{t,\bullet}(w)$. Thus, to account also for the changes in the shortest paths between w and the nodes in $T(w)$, $2\Delta_{s,\bullet}$ is subtracted from $c_B(w)$ in Line 4 (and analogously $2\Delta'_{s,\bullet}$ is added in the update of nodes in the new shortest paths).

6 Time complexity

Let us study the complexity of our two new algorithms for updating APSP and betweenness scores described in Section 4.3 and Section 5.3, respectively. We define the *extended size* $||A||$ of a set of nodes A as the sum of the number of nodes in A and the number of edges that have a node of A as their endpoint. Then, the following holds.

▶ **Theorem 3.** *The running time of Algorithm 1 for updating the augmented APSP after an edge insertion (or weight decrease)* $(u, v, \omega'(u, v))$ *is* $\Theta(||S(v)|| + ||T(u)|| + \sum_{y \in T(u)} |S(p(y))|)$*, where* $p(y)$ *can be any node in* $P_u(y)$*.*

Proof. The function `findAffectedSources` in Line 3 identifies the set of affected sources starting a BFS in v and visiting only the nodes $s \in S(v)$. This takes $\Theta(||S(v)||)$, since this pruned BFS visits all nodes in $S(v)$ and their incident edges. Then, the while loop of Lines 9 - 30 identifies all the affected targets $T(u)$ with a pruned BFS. This part (excluding Lines 11 - 22) requires $\Theta(||T(u)||)$ operations, since all affected targets and their incident edges are visited. In Lines 11 - 22, for each affected node $t \in T(u)$, all the affected sources of the predecessor $p(y)$ of y are scanned. This part requires in total $\Theta(\sum_{t \in T(u)} |S(p(y))|)$ operations. ◀

Notice that, since $|S(p(y))|$ is $O(n)$ and both $||T(u)||$ and $||S(v)||$ are $O(n+m)$, the worst-case complexity of Algorithm 1 is $O(n^2)$. To show the complexity of the dependency update described in Algorithm 2, let us introduce, for a given source node s, the set $\tau(s) := T(s) \cup \{w \in V : \Delta_{s,\bullet}(w) > 0\}$. Then, the following theorem holds.

▶ **Theorem 4.** *The running time of Algorithm 2 is* $\Theta(||\tau(s)|| + |\tau(s)| \log |\tau(s)|)$ *for weighted graphs and* $\Theta(||\tau(s)||)$ *for unweighted graphs.*

Proof. In the following, we assume a binary heap priority queue for weighted graphs and a bucket list priority queue for unweighted graphs. Then, the `extractMax()` operation in Line 3 requires constant time for unweighted and logarithmic time for weighted graphs. Also, for each node extracted from PQ, all neighbors are visited in Lines 5–18. Therefore, it is sufficient to prove that the set of nodes inserted into (and therefore extracted from) PQ is exactly $\tau(s)$. As we said in the description of Algorithm 2, PQ is initially populated with the nodes in $T(s)$. Then, all nodes y inserted into PQ in Line 14 are nodes that lied in at least one shortest path between s and a node in $T(s)$ before the insertion. This means that there is at least one $t \in T(s)$ such that $\sigma_{st}(y) > 0$, which implies that $\Delta_{s,\bullet}(y) > 0$, by definition of $\Delta_{s,\bullet}(y)$. ◀

The running time necessary to increase the betweenness score of nodes such that $\Delta'_{s,\bullet} > 0$ can be computed analogously, defining $\tau'(s) = T(s) \cup \{w \in V : \Delta'_{s,\bullet}(w) > 0\}$. Overall, the running time of the betweenness update score described in Section 5.3 is $\Theta(\sum_{s \in S} ||\tau(s)|| + ||\tau'(s)||)$ for unweighted and $\Theta(\sum_{s \in S} ||\tau(s)|| + ||\tau'(s)|| + |\tau(s)| \log |\tau(s)| + |\tau'(s)| \log |\tau'(s)|)$ for weighted graphs. Consequently, in the worst case, this is $O(nm)$ for unweighted and $O(n(m + n \log n))$ for weighted graphs, which matches the running time of BA. For sparse graphs, this is asymptotically faster than KWCC, which requires $\Theta(n^3)$ operations in the worst case.

7 Experimental Results

Implementation and settings. For our experiments, we implemented BA, KDB, KWCC, and our new approach, which we refer to as iBet (from Incremental Betweenness). All the algorithms were implemented in C++, building on the open-source *NetworKit* framework [24]. All codes are sequential; they were executed on a 64bit machine with 2 x 8 Intel(R) Xeon(R) E5-2680 cores at 2.7 GHz with 256 GB RAM with a single thread on a single CPU.

Data sets and experimental design. For our experiments, we consider a set of real-world networks belonging to different domains, taken from SNAP [17], KONECT [15], and LASAGNE (`piluc.dsi.unifi.it/lasagne`). Since KDB cannot handle weighted graphs and the pseudocode given in [14] is only for undirected graphs, all graphs used in the experiments are undirected and unweighted. The networks are reported in Table 1. Due to the time required by the static algorithm and the memory constraints of all dynamic algorithms ($\Theta(n^2)$), we only considered networks with up to about 26000 nodes.

To simulate real edge insertions, we remove an existing edge from the graph (chosen uniformly at random), compute betweenness on the graph without the edge and then re-insert the edge, updating betweenness with the incremental algorithms (and recomputing it with BA). For all networks, we consider 100 edge insertions and report the average over these 100 runs.

Table 1 The table shows the average time taken by the static algorithm BA and the average speedups on BA of the incremental algorithms (geometric means). The best result of each row is shown in bold font.

Graph	Nodes	Edges	Type	BA [s]	Speedup on BA iBet	KDB	KWCC
HC-BIOGRID	4 039	10 321	bio. network	6.06	**77.87**	10.91	18.33
Mus-musculus	4 610	5 747	bio. network	3.32	**119.23**	9.40	11.21
Caenor-elegans	4 723	9 842	metabolic	5.12	**130.89**	9.58	23.64
ca-GrQc	5 241	14 484	coauthorship	4.19	**206.55**	7.53	14.28
advogato	7 418	42 892	social	14.65	**295.39**	27.69	18.45
hprd-pp	9 465	37 039	bio. network	30.29	**304.24**	11.33	45.90
ca-HepTh	9 877	25 973	coauthorship	21.06	**199.04**	8.24	34.03
dr-melanogaster	10 625	40 781	bio. network	40.76	**235.54**	7.94	48.57
oregon1-010526	11 174	23 409	aut. systems	24.43	**237.47**	15.20	21.64
oregon2-010526	11 461	32 730	aut. systems	30.07	**113.10**	17.23	23.08
Homo-sapiens	13 690	61 130	bio. network	68.58	**237.61**	10.29	58.67
GoogleNw	15 763	148 585	hyperlinks	90.42	**577.49**	90.01	33.80
dip20090126	19 928	41 202	bio. network	115.56	**51.54**	5.38	5.73
as-caida20071105	26 475	53 381	aut. systems	154.36	**173.90**	18.66	19.65
Geometric mean					**179.1**	13.0	22.9

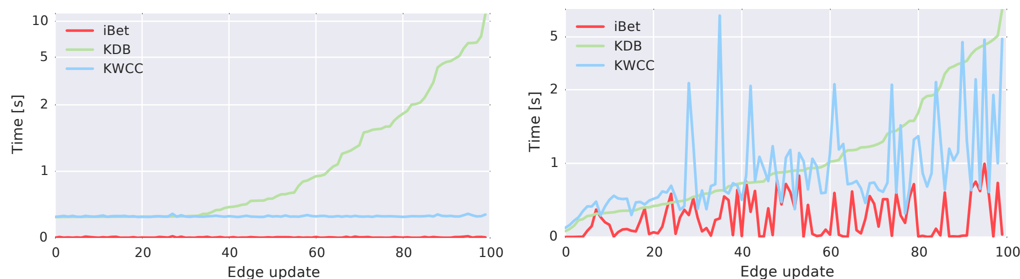


Figure 3 Running times of iBet, KDB and KWCC for 100 edge updates on `oregon1-010526`. Left: times for the APSP update step. Right: times for the dependency update step.

Experimental results. In Table 1 the running times of BA for each graph and the speedups of the three incremental algorithms on BA are reported. The last line shows the geometric mean of the speedups on BA over all tested networks. Our new method iBet clearly outperforms the other two approaches and is always faster than both of them. On average, iBet is faster than BA by a factor 179.1, whereas KDB by a factor 13.0 and KWCC by a factor 22.9.

Figure 3 compares the APSP update (on the left) and dependency update (on the right) steps for the `oregon1-010526` graph (a similar behavior was observed also for the other graphs of Table 1. On the left, the running time of the APSP update phase of the three incremental algorithms on 100 edge insertions are reported, sorted by the running time taken by KDB. It is clear that the APSP update of iBet is always faster than the competitors. This is due to the fact that iBet processes the edges between the affected targets only once instead of doing it once for each affected source as both KDB and KWCC. Also, the running time of the APSP update of KDB varies significantly. On about one third of the updates, it is basically as fast as KWCC. This means that in these cases, KDB only visits a small amount of nodes in addition to the affected ones (see Figure 1 and its explanation). However, in other cases KDB can be much slower, as shown in the figure.

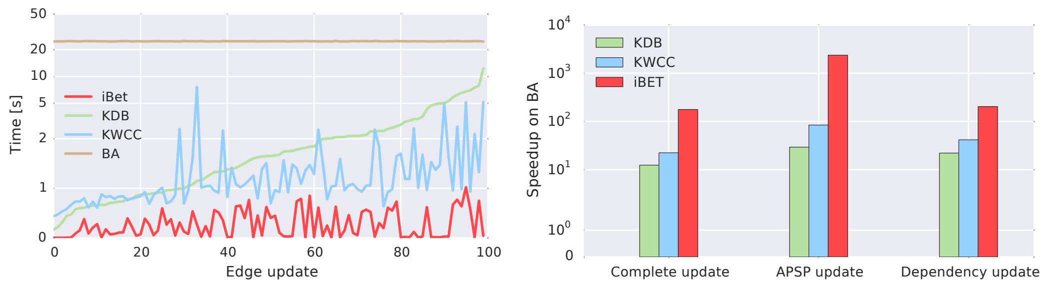


Figure 4 Left: Running times of iBet, KDB, KWCC and BA on the oregon1-010526 graph for 100 edge updates. Right: Average speedups on recomputation with BA (geometric mean) over all networks of Table 1 for the three incremental algorithms. The column on the left shows the speedup of the complete update, the one in the middle the speedup of the APSP update only and the one on the right the speedup of the dependency update only.

On the right of Figure 3, the running times of the dependency update step are reported. Also for this step, iBet is faster than both KDB and KWCC. However, for this part there is not a clear winner between KWCC and KDB. In fact, in some cases KDB needs to process additional nodes in order to recompute dependencies, whereas KWCC only processes nodes in the shortest paths between affected nodes. However, KDB processes each node at most once for each source node s, whereas KWCC might process the same node several times if it lies in several shortest paths between s and other nodes (we recall that the worst-case running time of KWCC is $O(n^3)$, whereas that of KDB is $O(nm)$). Notice also that in some rare cases KDB is slightly faster than iBet in the dependency update. This is probably due to the fact that our implementation of iBet is based on a priority queue, whereas KDB on a bucket list.

Figure 4 on the left reports the total running times of iBet, KDB, KWCC and BA on oregon1-010526. Although the running times vary significantly among the updates, iBet is always the fastest among all algorithms. On the contrary, there is not always a clear winner between KDB and KWCC. On the right, Figure 4 shows the geometric mean of the speedups on recomputation for the three incremental algorithms, considering the complete update, the APSP update step only and the dependency update step only, respectively. iBet is the method with the highest speedup both overall and on the APSP update and dependency update steps separately, meaning that each of the improvements described in Section 4.3 and Section 5.3 contribute to the final speedup. On average, iBet is a factor 82.7 faster than KDB and a factor 28.5 faster than KWCC on the APSP update step and it is a factor 9.4 faster than KDB and a factor 4.9 faster than KWCC on the dependency update step. Overall, the speedup of iBet on KDB ranges from 6.6 to 29.7 and is on average (geometric mean of the speedups) 14.7 times faster. The average speedup on KWCC is 7.4, ranging from a factor 4.1 to a factor 16.0.

8 Conclusions and future work

Computing betweenness centrality is a problem of great practical relevance. In this paper we have proposed and evaluated new techniques for the betweenness update after the insertion (or weight decrease) of an edge. Compared to other approaches, our new algorithm is easy to implement and significantly reduces the number of operations of both the APSP update and the dependency update. Our experiments on real-world networks show that our approach outperforms existing methods, on average approximately by one order of magnitude.

Future work might include parallelization for further acceleration. Furthermore, we plan to extend our techniques also to the decremental case (where an edge can be deleted from the graph or its weight can be increased) and to batch updates, where several edge updates might occur at the same time.

Although dynamic betweenness algorithms can be much faster than recomputation, a major limitation for their scalability is their memory requirement of $\Theta(n^2)$. An interesting research direction is the design of scalable dynamic algorithms with a smaller memory footprint.

Our implementations are based on *NetworKit* [24], the open-source framework for network analysis, and we will publish our source code in upcoming releases of the package.

References

1. David C. Bell, John S. Atkinson, and Jerry W. Carlson. Centrality measures for disease transmission networks. *Social Networks*, 21(1):1–21, 1999. doi:10.1016/S0378-8733(98)00010-0.
2. Elisabetta Bergamini and Henning Meyerhenke. Approximating betweenness centrality in fully dynamic networks. *Internet Mathematics*, 12(5):281–314, 2016. doi:10.1080/15427951.2016.1177802.
3. Elisabetta Bergamini, Henning Meyerhenke, and Christian Staudt. Approximating betweenness centrality in large evolving networks. In *17th Workshop on Algorithm Enginnering and Experiments, ALENEX 2015*, pages 133–146. SIAM, 2015. doi:10.1137/1.9781611973754.12.
4. Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014. doi:10.1080/15427951.2013.865686.
5. Michele Borassi and Emanuele Natale. KADABRA is an adaptive algorithm for betweenness via random approximation. In *24th Annual European Symposium on Algorithms, ESA 2016*, volume 57 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.20.
6. Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.
7. Pierluigi Crescenzi, Gianlorenzo D'Angelo, Lorenzo Severini, and Yllka Velaj. Greedily improving our own centrality in A network. In *Experimental Algorithms – 14th International Symposium, SEA 2015, Proceedings*, volume 9125 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2015. doi:10.1007/978-3-319-20086-6_4.
8. Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 90–100. SIAM, 2008.
9. Oded Green, Robert McColl, and David A. Bader. A fast algorithm for streaming betweenness centrality. In *SocialCom/PASSAT*, pages 11–20. IEEE, 2012.
10. Takanori Hayashi, Takuya Akiba, and Yuichi Yoshida. Fully dynamic betweenness centrality maintenance on massive networks. *Proceedings of 41st International Conference on Very Large Data Bases (PVLDB 2015)*, 9(2):48–59, 2015.
11. Miray Kas, Matthew Wachs, Kathleen M. Carley, and L. Richard Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *Advances in Social Networks Analysis and Mining 2013 (ASONAM'13)*, pages 33–40. ACM, 2013.
12. Christine Kiss and Martin Bichler. Identification of influencers – measuring influence in customer networks. *Decision Support Systems*, 46(1):233–253, 2008. doi:10.1016/j.dss.2008.06.007.

13 Dirk Koschützki, Katharina Anna Lehmann, Leon Peeters, Stefan Richter, Dagmar Tenfelde-Podehl, and Oliver Zlotowski. Centrality indices. In *Network Analysis*, volume 3418 of *LNCS*, pages 16–61. Springer Berlin Heidelberg, 2005. doi:10.1007/978-3-540-31955-9_3.

14 N. Kourtellis, G. De Francisci Morales, and F. Bonchi. Scalable online betweenness centrality in evolving graphs. *Knowledge and Data Engineering, IEEE Transactions on*, PP(99):1–1, 2015.

15 Jérôme Kunegis. KONECT: the koblenz network collection. In *22nd International World Wide Web Conference, WWW'13*, pages 1343–1350, 2013. URL: http://dl.acm.org/citation.cfm?id=2488173.

16 Min-Joong Lee, Sunghee Choi, and Chin-Wan Chung. Efficient algorithms for updating betweenness centrality in fully dynamic graphs. *Information Sciences*, 326:278–296, 2016.

17 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014. URL: http://snap.stanford.edu/data.

18 Chih-Chung Lin and Ruei-Chuan Chang. On the dynamic shortest path problem. *Journal of Information Processing*, 13(4):470–476, April 1991. URL: http://dl.acm.org/citation.cfm?id=105582.105591.

19 Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. Betweenness centrality – incremental and faster. In *Mathematical Foundations of Computer Science 2014 – 39th International Symposium, MFCS 2014*, volume 8635 of *Lecture Notes in Computer Science*, pages 577–588. Springer, 2014.

20 Matteo Pontecorvi and Vijaya Ramachandran. Fully dynamic betweenness centrality. In *Algorithms and Computation – 26th International Symposium, ISAAC 2015, Proceedings*, volume 9472 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2015.

21 G. Ramalingam and Thomas W. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1&2):233–277, 1996. doi:10.1016/0304-3975(95)00079-8.

22 Matteo Riondato and Evgenios M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, 30(2):438–475, 2016.

23 Matteo Riondato and Eli Upfal. ABRA: approximating betweenness centrality in static and dynamic graphs with rademacher averages. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016*, pages 1145–1154. ACM, 2016. doi:10.1145/2939672.2939770.

24 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A tool suite for high-performance network analysis. *Network Science*, To appear.

Fast Deterministic Selection

Andrei Alexandrescu

The D Language Foundation, Washington, USA

Abstract

The selection problem, in forms such as finding the median or choosing the k top ranked items in a dataset, is a core task in computing with numerous applications in fields as diverse as statistics, databases, Machine Learning, finance, biology, and graphics. The selection algorithm Median of Medians, although a landmark theoretical achievement, is seldom used in practice because it is slower than simple approaches based on sampling. The main contribution of this paper is a fast linear-time deterministic selection algorithm MedianOfNinthers based on a refined definition of MedianOfMedians. A complementary algorithm MedianOfExtrema is also proposed. These algorithms work together to solve the selection problem in guaranteed linear time, faster than state-of-the-art baselines, and without resorting to randomization, heuristics, or fallback approaches for pathological cases. We demonstrate results on uniformly distributed random numbers, typical low-entropy artificial datasets, and real-world data. Measurements are open-sourced alongside the implementation at `https://github.com/andralex/MedianOfNinthers`.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Selection Problem, Quickselect, Median of Medians, Algorithm Engineering, Algorithmic Libraries

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.24

1 Introduction

The selection problem is widely researched and has numerous applications. Selection is finding the kth smallest element (also known as the kth order statistic): given an array A of length $|A| = n$, a non-strict order $\leq$ over elements of A, and an index k, the task is to find the element that would be in slot $A[k]$ if A were sorted. A variant that is the focus of this paper is *partition-based selection*: the algorithm must also permute elements of A such that $A[i] \leq A[k]\ \forall i, 0 \leq i < k$, and $A[k] \leq A[i]\ \forall i, k \leq i < n$.

Quickselect, originally called Find by its creator C.A.R. Hoare [17], is the selection algorithm most used in practice [29, 8, 25]. Like Quicksort [16], Quickselect relies on a separate routine Partition to divide the array into elements less than or equal to, and greater than or equal to, a specifically chosen element called the pivot. Unlike Quicksort which recurses on both subarrays left and right of the pivot, Quickselect only recurses on the side known to contain the kth smallest element.

The pivot choosing strategy is crucial for Quickselect because it conditions its performance between $O(n)$ and $O(n^2)$. Commonly used heuristics for pivot choosing – e.g. the median of 1–9 elements [30, 14, 3, 8] – work well on average but have high variance and do not offer worst-case guarantees. The "Median of Medians" pivot selection algorithm [4] solves the selection problem in guaranteed linear time. However, MedianOfMedians is seldom used in practice because it is intensive in element comparisons and especially swaps. Musser's Introselect algorithm [25] proceeds with a heuristics-informed Quickselect that monitors its own performance and only switches to MedianOfMedians if progress is slow. Contemporary implementations of selection (such as GNU C++ STL [13] and NumPy [28])

© Andrei Alexandrescu;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 24; pp. 24:1–24:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

use Quickselect or Introselect in conjunction with simple pivot choosing heuristics. However, all current implementations are prone to high variance in run times even if we discount rare pathological cases. If heuristics provide poor pivot choices for the first 1–4 partition passes (when there is most data to process), the total run time increases strongly above its average [18, 9]. A selection algorithm that combines the principled nature and theoretical guarantees of MedianOfMedians with good practical performance on average has remained elusive.

We seek to improve the state of the art in deterministic selection. First, we improve the definition of MedianOfMedians to reduce comparisons and swaps. Second, we introduce *sampling* to improve average performance without compromising algorithm's linear asymptotic behavior. The resulting MedianOfNinthers algorithm is principled, practical, and competitive. Third, we introduce *adaptation* based on the observation that MedianOfMedians is specialized to find the median, but may be used with any order statistic. That makes its performance degrade considerably when selecting order statistics away from the median. (These situations occur naturally even when searching for the median due to the way Quickselect works.) We devise a simple and efficient partitioning algorithm MedianOfExtrema for searching for order statistics close to either end of the searched array. A driver algorithm QuickselectAdaptive chooses dynamically the most appropriate partitioning method to find the median in linear time without resorting to randomization, heuristics, or fallback approaches for pathological cases. Most importantly, QuickselectAdaptive does not compromise on efficiency – it was measured to be faster than a number of baselines, notably including GNU C++ `std::nth_element`. We open-sourced the implementation of QuickselectAdaptive along with the benchmarks and datasets used in this paper [1].

The paper uses the customary pseudocode and algebraic notations. Divisions of integrals yield the floor as in many programming languages, e.g. $n/3$ or $\frac{n}{3}$ are $\lfloor \frac{n}{3} \rfloor$. We make the floor notation explicit when at least one operand is not integral. Arrays are zero-based. The length of array A is written as $|A|$. We denote with $A[a : b]$ (if $a < b$) a "slice" of A starting with $A[a]$ and ending with (and including) $A[b - 1]$. The slice is empty if $a = b$. Elements of the array are not copied – the slice is a bounded view into the given array.

2 Related Work

Hoare created the Quickselect algorithm in 1961 [17], which still is the preferred choice of practical implementations, in conjunction with various pivot choosing strategies. Martínez et al. [22] analyze the behavior of Quickselect with small data sets and propose stopping Quickselect's recursion early and using sorting as an alternative policy below a cutoff, essentially a simple multi-strategy Quickselect. The same authors [23] propose several adaptive sampling strategies for Quickselect that take into account the index searched.

Blum, Floyd, Pratt, Rivest, and Tarjan created the seminal MedianOfMedians algorithm [4], also known as BFPRT from its authors' initials. Subsequent work provided variants and improved on its theoretical bounds [12, 33, 34, 21, 5, 11]. Chen and Dumitrescu [6] propose RepeatedStep (discussed in detail in §3), a variant of MedianOfMedians that groups 3 or 4 elements (the original uses groups of 5 or more elements) and prove its linearity. Battiato et al. [2] describe Sicilian Median Selection, an algorithm for computing an approximate median that may be considered the transitive closure of RepeatedStep.

Floyd and Rivest created the randomized SELECT algorithm [12] in 1975. Although further improved and benchmarked with favorable results by Kiwiel [19], at the time of this writing we found no implementation available online and no evidence of industry adoption.

Algorithm 1: QUICKSELECT.

Data: PARTITION, A, k with $|A| > 0, 0 \leq k < |A|$
Result: Puts kth smallest element of A in $A[k]$ and partitions A around it.

```
1  while true do
2      p ← Partition(A);
3      if p = k then
4          return;
5      end
6      if p > k then
7          A ← A[0 : p];
8      else
9          k ← k − p − 1;
10         A ← A[p + 1 : |A|];
11     end
12 end
```

Algorithm 2: HOAREPARTITION.

Data: A, p with $|A| > 0, 0 \leq p < |A|$
Result: p', the new position of $A[p]$; A partitioned at $A[p']$

```
1  Swap(A[p], A[0]);
2  a ← 1;
3  b ← |A| − 1;
4  loop: while true do
5      while true do
6          if a > b then break loop;
7          if A[a] ≥ A[0] then break;
8          a ← a + 1;
9      end
10     while A[0] < A[b] do b ← b − 1;
11     if a ≥ b then break;
12     Swap(A[a], A[b]);
13     a ← a + 1;
14     b ← b − 1;
15 end
16 Swap(A[0], A[a − 1]);
17 return a − 1;
```

3 Background: Quickselect, MedianOfMedians, and RepeatedStep

QUICKSELECT [8, 20] takes as parameters the input array A and the sought order statistic k. To facilitate exposition, our variant (Algorithm 1) also takes as parameter the partitioning primitive PARTITION, in a higher-order function fashion. PARTITION(A) chooses and returns an index $p \in \{0, 1, \ldots, |A|-1\}$ called *pivot*, and also partitions A around $A[p]$. QUICKSELECT uses the pivot to either reduce the portion of the array searched from the left when $p < k$, reduce it from the right when $p > k$, or end the search when $p = k$.

The running time of QUICKSELECT is linear if PARTITION(A) returns in linear time a pivot p ranked within a fixed fraction $0 < f < 1$ from either extremum of A. Most pivot selection schemes use *heuristics* by choosing a pivot unlikely to be close to an extremum in conjunction with HOAREPARTITION, which partitions the array in $O(n)$. (Many variants of Hoare's PARTITION algorithm [15] exist; Algorithm 2 is closer to the implementation we used, than to the original definition.) Heuristics cannot provide a good worst-case run time guarantee for QUICKSELECT, but perform well on average.

MEDIANOFMEDIANS, prevalently implemented as shown in BFPRTBASELINE (Algorithm 3) [10, 7, 26, 32], spends more time to guarantee good pivot choices. The algorithm first computes medians of groups of 5 elements for a total of $\frac{|A|}{5}$ groups. The rote routine MEDIAN5(A, a, b, c, d, e) swaps the median of $A[a], \ldots, A[e]$ into $A[c]$. Computing the median of these medians yields a pivot p with a useful property. There are $\frac{|A|}{10}$ elements less than or equal to $A[p]$, but each of those is the median of 5 distinct elements, so $A[p]$ is not smaller than at least $\frac{3|A|}{10}$ elements. By symmetry, $A[p]$ is not greater than at least $\frac{3|A|}{10}$ elements.

Algorithm 3: BFPRTBaseline.

```
   Data: A
   Result: Pivot 0 ≤ p < |A|; A partitioned at A[p]
 1 if |A| < 5 then
 2 |  return HoarePartition(A, |A|/2);
 3 end
 4 i ← 0;  j ← 0;
 5 while i + 4 < |A| do
 6 |  Median5(A, i, i + 1, i + 2, i + 3, i + 4);
 7 |  Swap(A[i + 2], A[j]);
 8 |  i ← i + 5;
 9 |  j ← j + 1;
10 end
11 Quickselect(BFPRTBaseline, A[0 : j], j/2);
12 return HoarePartition(A, j/2);
```

Algorithm 4: RepeatedStep.

```
   Data: A
   Result: Pivot 0 ≤ p < |A|; A partitioned at A[p]
 1 if |A| < 9 then
 2 |  return HoarePartition(A, |A|/2);
 3 end
 4 i ← 0;  j ← 0;
 5 while i + 2 < |A| do
 6 |  Median3(A, i, i + 1, i + 2);
 7 |  Swap(A[i + 1], A[j]);
 8 |  i ← i + 3;
 9 |  j ← j + 1;
10 end
11 i ← 0;  m ← 0;
12 while i + 2 < j do
13 |  Median3(A, i, i + 1, i + 2);
14 |  Swap(A[i + 1], A[m]);
15 |  i ← i + 3;
16 |  m ← m + 1;
17 end
18 Quickselect(RepeatedStep, A[0 : m], m/2);
19 return HoarePartition(A, m/2);
```

Selection is initiated by invoking Quickselect(BFPRTBaseline, A, k). To prove linearity, let us look at the worst-case number of comparisons depending on $n = |A|$:

$$C(n) \leq C\left(\frac{n}{5}\right) + C\left(\frac{7n}{10}\right) + \frac{6n}{5} + n \tag{1}$$

where the first term accounts for the median of medians computation, the second is the time taken by Quickselect after partitioning, the third is the cost of computing the medians of five, and the last is the cost of HoarePartition. Consequently $C(n) \leq 22n$.

The number of swaps is also of interest. BFPRTBaseline uses a common optimization [10, 7, 26, 32] – it reuses the first quintile of A for storing the medians. $S(n)$ satisfies:

$$S(n) \leq S\left(\frac{n}{5}\right) + S\left(\frac{7n}{10}\right) + \frac{7n}{5} + \frac{n - \frac{n}{10}}{2}\,. \tag{2}$$

The terms correspond to those for $C(n)$. Consequently $S(n) \leq \frac{37n}{2}$. However, neither bound is tight, meaning they only have advisory value; given that generating worst-case data for MedianOfMedians remains an open problem, we use empirical benchmarks (§ 7) to compare the performance of all algorithms discussed.

Recently Chen and Dumitrescu [6] proposed linear-time MedianOfMedians variants that use groups of 3 or 4 elements, disproving long-standing conjectures to the contrary. Algorithm 4 shows the pseudocode of their RepeatedStep algorithm with a group size of 3.

Key to the algorithm is that the median of medians step is repeated, thus choosing the pivot as the median of medians of medians of three (sic). This degrades the pivot's quality, placing it within $\frac{2n}{9}$ elements from either extremum of A. However, the median of medians computation only needs to recurse on $\frac{n}{9}$ elements. Intuitively, trading off some pivot quality for faster processing in MedianOfMedians is a good idea for competing with imprecise but fast pivot heuristics. $C(n)$ for Quickselect(RepeatedStep, A, k) satisfies ($n = |A|$):

$$C(n) \leq C\left(\frac{n}{9}\right) + C\left(\frac{7n}{9}\right) + \frac{3n}{3} + \frac{3n}{9} + n \tag{3}$$

where the first term is the cost of computing the median of medians of medians, the second is the worst-case time spent processing the remaining elements, and the last three terms account respectively for computing the medians of 3, computing the medians of 3 medians of 3, and the partitioning. Consequently $C(n) \le 21n$. For $S(n)$, each median of three uses at most 1 swap, to which we add 1 for swapping the median to the front of the array. Partitioning costs at most $\frac{n}{2}$ swaps in general, but the first $\frac{n}{18}$ elements are not swapped:

$$S(n) \le S\left(\frac{n}{9}\right) + S\left(\frac{7n}{9}\right) + \frac{2n}{3} + \frac{2n}{9} + \frac{n - \frac{n}{18}}{2} \tag{4}$$

which solves to $S(n) \le \frac{49n}{4}$. These bounds prove linearity but are not necessarily tight.

4 Partitioning During Pivot Computation

We now set out to improve these algorithms. One starting observation is that heuristics-based partition uses a $O(1)$ step (picking a pivot) followed by a linear pass (invoking HOAREPARTITION). In contrast, BFPRTBASELINE and REPEATEDSTEP make *two* linear passes: one for finding the pivot, and one for the partitioning step (also using HOAREPARTITION). Therefore, algorithms in the MEDIANOFMEDIANS family are at a speed disadvantage.

This motivates one key insight: we aim to *integrate* the two steps, i.e. make the comparisons and swaps performed during pivot computation also count toward partitioning. REPEATEDSTEP organizes the array in groups of 3 and computes the median of each group; then it repeats the same procedure for the medians of three. That imparts a non-trivial implicit structure onto the input array in addition to computing the pivot. However, that structure is not used by HOAREPARTITION. Ideally, that structure should be embedded in the array in a form favorable to the subsequent partitioning step.

Our approach (MEDIANOFNINTHERSBASIC shown in Algorithm 5) is to make the small groups *non-contiguous* and choose their placement in a manner that is advantageous for the partitioning step, so as to avoid comparing and swapping elements more than once. To that end, we place the subarray of medians in the very middle of A, more precisely in the 5th 9^{th}ile of the array. (Recall that REPEATEDSTEP computes a subarray of medians of medians with $\frac{|A|}{9}$ elements and recurses against it to compute its median.)

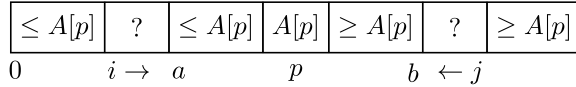
Also, instead of executing two loops, we execute a single pass that ensures the same postcondition. This is done with Tukey's NINTHER routine [31, 3], which takes 9 array elements, computes the medians of 3 disjoint groups of 3, and yields the median of those 3 medians. Specifically, NINTHER$(A, i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9)$ computes the index of the median of $A[i_1]$, $A[i_2]$, and $A[i_3]$ into i'_1, the index of the median of $A[i_4]$, $A[i_5]$, and $A[i_6]$ into i'_2, and the index of the median of $A[i_7]$, $A[i_8]$, and $A[i_9]$ into i'_3. Then it swaps $A[i_5]$ with the median of $A[i'_1]$, $A[i'_2]$, and $A[i'_3]$. After this operation, $A[i_5]$ is no less than at least 3 elements and no greater than at least 3 other elements among $A[i_1]$, $\dots$, $A[i_9]$. We use NINTHER against groups that pick 4 elements from the front of A, one from the mid 9^{th}ile (which receives the median), and 4 from the right of that 9^{th}ile.

These changes have important theoretical and practical advantages. First, NINTHER computes the same medians of 3 medians of 3 as the first two loops in REPEATEDSTEP using the same number of comparisons (12 per group of 9) but with 0–1 swaps instead of 0–3. Second, a single pass is better than the two successive loops in REPEATEDSTEP. Third, after recursing to QUICKSELECT against $A\left[\frac{4|A|}{9} : \frac{5|A|}{9}\right]$, the mid 9^{th}ile is already partitioned properly around the pivot; there is no need to visit it again. That way, the medians computation step contributes one ninth of the final result at no additional cost.

Algorithm 5: MEDIANOFNINTHERSBASIC.

```
   Data: A
   Result: Pivot p, 0 ≤ p < |A|; A partitioned at p
 1 if |A| < 9 then
 2 |  return HOAREPARTITION(A, |A|/2);
 3 end
 4 f ← |A|/9;
 5 for i ← 4f through 5f − 1 do
 6 |  l ← i − 4f;
 7 |  r ← i + 5f;
 8 |  NINTHER(A, l, l + 1, l + 2, l + 3, i,
   |    r, r + 1, r + 2, r + 3);
 9 end
10 QUICKSELECT(REPEATEDSTEPBASIC,
     A[4f : 5f], f/2);
11 return
     EXPANDPARTITION(A, 4f, 4f + f/2, 5f − 1);
```

The pivot computation leaves the array well suited for partitioning by visiting 9^{th}iles 1–4 and 6–9 (subarrays $A\left[0 : \frac{4|A|}{9}\right]$ and $A\left[\frac{5|A|}{9} : |A|\right]$). This work is carried by EXPANDPARTITION (not shown, available online [1]), a modified HOAREPARTITION algorithm that takes into account the already-partitioned subarray around the pivot. The call EXPANDPARTITION(A, a, p, b) proceeds by the following scheme, starting with $i = 0$ and $j = |A| - 1$ and moving them toward a and b, respectively:



The procedure swaps as many elements as possible between $A[i : a]$ and $A[b + 1 : j + 1]$ because that is the most efficient use of swapping – one swap puts two elements in their final slots. There may be some asymmetry (one of i and j reaches its limit before the other) so the pivot position may shift to the left or right. EXPANDPARTITION returns the final position of the pivot, which BFPRTIMPROVED forwards to the caller. EXPANDPARTITION(A, a, b) performs $a + |A| - b$ comparisons and at most $\max(a, |A| - b)$ swaps.

The number of comparisons $C(n)$ satisfies the recurrence:

$$C(n) \le C\left(\frac{n}{9}\right) + C\left(\frac{7n}{9}\right) + n + \frac{n}{3} + \frac{8n}{9}\,. \tag{5}$$

The only difference from the corresponding bound of REPEATEDSTEP is the last term, which is slightly smaller in this case because we don't revisit the middle 9^{th}ile during partitioning. The recurrence solves to $C(n) \le 20n$.

For $S(n)$, each NINTHER contributes at most 1 swap per 9 elements. In the worst case EXPANDPARTITION needs to swap $\frac{4n}{9}$ elements from the left side with $\frac{4n}{9}$ elements from the right. However, the swaps don't sum because one swap operation takes care of two elements wherever possible. So the number of swaps performed by EXPANDPARTITION is at most $\frac{4n}{9}$.

$$S(n) \le S\left(\frac{n}{9}\right) + S\left(\frac{7n}{9}\right) + \frac{n}{9} + \frac{4n}{9}\,. \tag{6}$$

Consequently $S(n) \le 5n$, a sizeable improvement over REPEATEDSTEP. Integrating pivot searching with partitioning is crucial for improving efficiency. MEDIANOFNINTHERSBASIC not only has much better performance, but also allows further optimizations to build upon it.

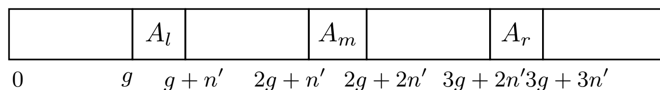
5 Sampling Without Compromising Linearity

We are now ready to introduce MEDIANOFNINTHERS (Algorithm 6). It uses a hyperparameter $0 < \phi \le 1$ from which it derives a subsample size $n' = \left\lfloor \frac{\phi|A|}{3} \right\rfloor$ and a gap length $g =$

Algorithm 6: MedianOfNinthers.

```
   Data: A
   Result: Pivot p, 0 ≤ p < |A|; A partitioned at p
 1 n ← |A|;
 2 n' ← ⌊φn/3⌋;
 3 if n' < 3 then
 4 |  return HoarePartition(A, |A|/2);
 5 end
 6 g ← (n − 3n')/4;
 7 A_m ← A[2g + n' : 2g + 2n'];
 8 l ← g;
 9 m ← 2g + n';
10 r ← 3g + 2n';
11 for i ← 0 through n'/3 − 1 do
12 |  Ninther(A, l, m, r, l + 1, m + n'/3, r + 1,
   |    l + 2, m + 2n'/3, r + 2, m, m + n'/3,
   |    m + 2n'/3);
13 |  m ← m + 1;
14 |  l ← l + 3;
15 |  r ← r + 3;
16 end
17 Quickselect(MedianOfNinthers, A_m, n'/2);
18 return ExpandPartition(A,
      2g + n', 2g + n' + n'/2, 2g + 2n');
```

$\frac{|A|-3n'}{4}$, after which it chooses three equidistant disjoint subarrays from A as follows: $A_l = A[g : g+n']$, $A_m = A[2g+n' : 2g+2n']$, and $A_r = A[3g+2n' : 3g+3n']$. The three subarrays have length n' each, and the gaps around them have length g each (except for the last gap which is longer by $(|A|-3n') \mod 4$), as illustrated below.



The plan is to save time by computing the pivot solely by looking at A_l, A_m, and A_r instead of visiting the entire array. The approach is essentially to perform the same algorithm as MedianOfNinthersBasic against the conceptual concatenation of A_l, A_m, and A_r.

The challenge is choosing an appropriate iteration schedule for picking the 9 elements to pass to Ninther. We do so by pairing successive triples of adjacent elements in A_l with the triple $A_m[i]$, $A_m[i+n'/3]$, $A_m[i+2n'/3]$ (having i range in $\{0, 1, \ldots, n'/3-1\}$) and with successive triples of adjacent elements in A_r. The result of each ninther is swapped to $A_m[2i]$, i.e. to the second tertile of A_m, which in turn is in the middle of A. After the loop, the second tertile of A_m contains the medians needed for recursion.

Let us prove linearity of Quickselect(MedianOfNinthers, A, k) by computing an upper bound of the number of comparisons $C(n)$ from the recurrence:

$$C(n) \leq C\left(\frac{\phi n}{9}\right) + C\left(n - \frac{2\phi n}{9}\right) + \frac{12\phi n}{9} + \frac{n(9-\phi)}{9}. \tag{7}$$

The first term accounts for the recursive call to Quickselect, which processes that many elements. The second term accounts for processing the remainder of the array (as reasoned for RepeatedStep, in the worst case $\frac{2\phi n}{9}$ elements are eliminated in one partitioning step), the third is the cost of the first loop (12 comparisons for each Ninther call, of which there are $\frac{\phi n}{9}$), and the last is the cost of ExpandPartition. Consequently $C(n) \leq \frac{(11\phi+9)n}{\phi}$. As expected, the number of comparisons goes up as ϕ goes down. For swaps we obtain (with the same term positions):

$$S(n) \leq S\left(\frac{\phi n}{9}\right) + S\left(n - \frac{2\phi n}{9}\right) + \frac{\phi n}{9} + \frac{(9-\phi)n}{18} \tag{8}$$

resulting in the bound $S(n) \leq \frac{(\phi+9)n}{2\phi}$. Although this result is theoretically unremarkable, it is attractive engineering-wise. It means we can fine-tune the tradeoff between the time spent computing the pivot and the quality of the pivot, without ever losing the linearity of the

algorithm. An entire spectrum opens up between the constant sample size used by heuristics and the full scan performed by all MedianOfMedians variations discussed so far.

6 Adaptation: MedianOfExtrema

Finding the kth order statistic in A is most difficult for $k = \frac{|A|}{2}$. However, sometimes k may be closer to one side of A than to its middle. Skewed values of k are possible not only when requested by the caller (e.g. fetch the 1000 best-ranked items from a large input), but also while computing the median proper. Recall that Quickselect (Algorithm 1) reduces $|A|$ progressively in a loop, which changes the relationship between $|A|$ and k. A few iterations bring the median search to an endgame of chasing a k close to 0 or $|A|$.

Quickselect(A, k) runs faster for skewed values of k than for $k = \frac{|A|}{2}$ because any pivot choosing method has a higher likelihood of finding a good pivot (one that allows eliminating a large fraction of the searched array). This puts elaborate pivot computing methods at a disadvantage compared to simple heuristics, so such situations are worth addressing. To that end we define a specialized algorithm MedianOfExtrema with two variants, MedianOfMinima for order statistics close to 0, and MedianOfMaxima for order statistics close to $|A|$. In the following we limit the discussion to MedianOfMinima. The corresponding variant MedianOfMaxima is defined analogously.

For small values of k relative to $|A|$, the partition function should *not* find a pivot to the left of k because that would only eliminate a small portion of the input. Martínez et al. discuss this risk for their related proportional-of-3 strategy [23]. So we require that MedianOfMinima must find a pivot not smaller than k.

MedianOfMinima (Algorithm 7) is based on the following intuition. MedianOfMedians computes medians of small groups and takes their median to find a pivot approximating the median of A. In this case, we pursue an order statistic skewed to the left, so instead of the median of each group, we compute its *minimum*; then, we obtain the pivot by computing the median of those groupwise minima. By construction, the pivot's rank will be shifted to the left of the true median. This is an easier task, too, because computing the minimum of a group is simpler and computationally cheaper than computing the group's median. We place these minima at A's front so they don't need to be swapped again.

The outer loop in Algorithm 7 organizes A such that its first $2k$ slots contain the minima of groups of size $\gamma = \frac{|A|}{2k}$ elements. Specifically, $A[0]$ receives the minimum over $A[0]$ and the first group of $\gamma - 1$ elements of $A\,[2k : |A|]$; $A[1]$ receives the minimum over $A[1]$ and the second group over $\gamma - 1$ elements of $A\,[2k : |A|]$; and so on through $A[2k-1]$, which receives the minimum over $A[2k-1]$ and the $2k^{\text{th}}$ group of $\gamma - 1$ elements of $A\,[2k : |A|]$. This permutation ensures that for each element in $A\,[0 : 2k]$, there are at least $\gamma - 1$ additional elements in $A\,[2k : |A|]$ greater than or equal to it.

The next steps compute the kth order statistic over $A\,[0 : 2k]$ (i.e. the upper median of the subarray of minima, as the name of the algorithm suggests), and uses the obtained $A[k]$ as pivot to expand the obtained partition to the entire array A. The recursive call replaces Quickselect with QuickselectAdaptive. The latter (fully specified in the next section) chooses to use either MedianOfMinima, MedianOfNinthers, or MedianOfMaxima depending on the ratio of k to $|A|$.

Let us assess the quality of the pivot p obtained by calling MedianOfMinima(A, k), i.e. the length of the subarray we can eliminate from the search after one call to MedianOfMinima. Obviously $p \geq k$ because the recursive call to QuickselectAdaptive places k elements to the left of $A[k]$ that are no greater than it. In addition, each of the k

Algorithm 7: MedianOfMinima.

```
   Data: A, 0 < k < |A|/6
   Result: Pivot p, k ≤ p ≤ |A|/2; A partitioned at A[p]
 1 if |A| = 1 then
 2 |  return 0;
 3 end
 4 γ ← |A|/2k;
 5 k ← G;
 6 for i ← 0 through 2k − 1 do
 7 |  m ← 2k + i(γ − 1);
 8 |  for j ← m + 1 through m + γ − 1 do
 9 |  |  if A[j] < A[j - 1] then
10 |  |  |  m ← j;
11 |  |  end
12 |  end
13 |  if A[m] < A[i] then
14 |  |  Swap(A[i], A[m]);
15 |  end
16 end
17 QuickselectAdaptive(A[0 : 2k], k);
18 return ExpandPartition(A, 0, k, 2k);
```

elements in subarray $A[k : 2k]$ is greater than or equal to the pivot; but by construction, for each of these elements there are $\gamma - 1$ others greater than or equal to the pivot in the subarray $A[2k : |A|]$. It follows that at least $k \left\lfloor \frac{|A|}{2k} \right\rfloor$ elements of A are greater than or equal to the pivot. Through algebraic manipulation we get $k \left\lfloor \frac{|A|}{2k} \right\rfloor \geq \frac{|A|-2k+1}{2} \geq \frac{|A|}{2} - k$ so the call MedianOfMinima(A, k) yields a pivot that allows the elimination of at least $\frac{|A|}{2} - k$ elements from the search.

These elements are eliminated from the search at the next iteration of QuickselectAdaptive, and we want to make sure the cost of the computation stays within linear bounds. The cost of eliminating these elements is $n - k$ for the minima computations plus a recursion on $2k$ elements. We conservatively require by the Master Theorem [27] that the recursion on $2k$ elements eliminates more than $2k$ elements, so $\frac{n}{2} - k > 2k$, which results in the requirement $k < \frac{n}{6}$. Conversely, for the MedianOfMaxima the threshold is $k > \frac{5n}{6}$. These thresholds work well in practice and are used in the implementation and experiments.

6.1 Choosing Strategy Dynamically: QuickselectAdaptive

In order to implement adaptation, we need to dynamically choose the partitioning algorithm from among MedianOfMinima, MedianOfNinthers, and MedianOfMaxima. A good place to decide strategy is the Quickselect routine itself, which has access to the information needed and drives the selection process. Before each partitioning step, the appropriate partitioning algorithm is chosen depending on the relationship between $|A|$ and k. After partitioning, both A and k are modified and a new decision is made, until the search is over. QuickselectAdaptive (Algorithm 8) embodies this idea.

7 Experiments and Results

For the implementation [1] we choose the sampling constant for MedianOfNinthers $\phi = \frac{1.0}{64.0}$ for arrays up to 2^{17} elements and $\phi = \frac{1.0}{1024.0}$ for larger arrays. Performance is not highly dependent on ϕ, for example there are no dramatic changes when halving or doubling ϕ. However, sampling is needed; with $\phi = 1$, the algorithm falls behind the best baseline. The data sets used are:

- *random*: uniformly-distributed random floating-point numbers.
- *random01*: $\frac{n}{2}$ zeros and $\frac{n}{2}$ ones, shuffled randomly. This puts to test algorithms' ability to cope with many duplicates.

Algorithm 8: QuickselectAdaptive.

Data: A, k with $0 \le k < |A|$
Result: Puts kth smallest element of A in $A[k]$ and partitions A around it.

```
1  while true do
2      if |A| ≤ 16 then
3          p ← HoarePartition(A, k);
4      else if 6k < |A| then
5          p ← MedianOfMinima(A, k);
6      else if 6k > 5|A| then
7          p ← MedianOfMaxima(A, k);
8      else
9          p ← MedianOfNinthers(A);
10     end
11     if p = k then return;
12     if p > k then
13         A ← A[0 : p];
14     else
15         i ← k − p − 1;
16         A ← A[p + 1 : |A|];
17     end
18 end
```

- *m3killer*: Musser's "median-of-3-killer sequence" [25].
- *organpipe*: numbers $0, 1, 2, \ldots, \frac{n}{2} - 1, \frac{n}{2} - 1, \ldots, 1, 0$.
- *sorted*: numbers $0, 1, 2, \ldots, n - 1$.
- *rotated*: numbers $1, 2, \ldots, n - 1, 0$.
- *googlebooks*: We complement artificial data sets with a real-world task – compute the median frequency of 1-grams (words) in different languages in the Google Ngrams corpus [24]. These data sets consist of between $5.4M$ and $20M$ 1-grams along with their frequencies. Words have been grouped per year with summing of frequencies. The part-of-speech annotations have been kept. The languages processed are English (eng), Russian (rus), French (fre), German (ger), Italian (ita), and Spanish (spa).

The artificial data sizes increase exponentially from $10,000$ to $10,000,000$ with step $\sqrt{10}$.

For baseline algorithms, we chose the pivot strategies most competitive and in prevalent industrial use: MedianOf3Randomized (which chooses the pivot as the median of three random array elements), Ninther (Tukey's ninther deterministic), and GNUIntroselect, GNU's implementation of the C++ standard library function `std::nth_element`. (To avoid clutter, we did not plot other heuristics that performed worse, such as single random pivot, ninther randomized, and median of 3 and 5 elements.)

First, we benchmarked run times on a desktop computer (Intel Core i7 3.6 GHz) against arrays of 64-bit floating point data. The compiler used was `gcc` version 5.4.0 invoked with `-O4 -DNDEBUG`. We ran each experiment 102 times, eliminated the 2 longest measurements to account for outliers caused by cache warmup and other additive noise, and took the average of the remaining timings. For random data, the input was randomly shuffled before each run.

GNUIntroselect is our main baseline because it is an independent and mature implementation that has received extensive use and scrutiny. All speed benchmarks plotted are normalized such that GNUIntroselect has relative speed $y = 1.0$, so as to make it easier to compare algorithms across widely different input sizes. Larger numbers are better, e.g. $y = 2.0$ means twice the speed.

Figure 1 plots the run times of the algorithms tested for finding the median in arrays of uniformly-distributed floating point numbers. (All run times are given in Appendix A.) QuickselectAdaptive is faster by a large margin for all data sizes.

For the *random01* data set (Figure 2), again the ranking puts QuickselectAdaptive first (albeit by a smaller margin), followed by GNUIntroselect. There is no noticeable difference between the performances of the two other algorithms.

The *m3killer* dataset (Figure 3) has GNUIntroselect as winner for most data sizes. The reason is that the median-of-3-killer pattern was intended to cause quadratic behavior to algorithms choosing the pivot as the median of $A[0]$, $A[|A|/2]$, and $A[|A|-1]$, but GNUIntroselect uses $A[1]$ instead of $A[0]$. Therefore, the first pivot chosen by GNUIntroselect is the median of $|A|/2+1$, 2, and $|A|$, which is exactly the upper median of the entire array. Therefore, all other algorithms compete against one single pass through a highly optimized implementation of HoarePartition.

The *organpipe* dataset (Figure 4) features Ninther and QuickselectAdaptive as best performers. Ninther makes good median choices because its sample positions are close to the actual median (which is at the 25th and 75th percentiles). QuickselectAdaptive's sampling strategy also finds the median with relative ease. The same pattern can be noted on the *sorted* dataset (Figure 5).

Figure 6 shows one interesting pathological case: GNUIntroselect is up to 30x slower than the other algorithms, and the gap grows with the size of the data set. This is surprising because the *rotated* data set (essentially a sorted sequence with a small value at the end) may plausibly occur in practice (e.g. a sorted array with one appended element).

Figure 7 compares performance for computing the median frequency of words in the Google Ngrams corpus [24]. QuickselectAdaptive outperforms all baselines.

7.1 Measuring comparisons, swaps, and variance of run times

Next, we tested the hypothesis (made in the introduction) that QuickselectAdaptive has lower variance than heuristics-based algorithms, by measuring and comparing the coefficient of variation $\frac{\sigma}{\mu}$ (standard deviation divided by mean) of run times. (Comparing σ values directly would not be appropriate because they characterize distributions with different averages.) We also measured the number of comparisons and swaps. The worst-case number of comparisons for computing the median has the lower bound $C(n) = (2+\epsilon)n$, where $\epsilon > 0$ is a constant [11]. On random data, the expected number of swaps by an optimal median selection algorithm is $S(n) = \frac{n}{4}$ (statistically half of the elements on either side of the median need to be swapped).

For the algorithms tested, Figure 8 shows comparisons per element $\frac{C(n)}{n}$, Figure 9 shows swaps per element $\frac{S(n)}{n}$, and Figure 12 shows the coefficient of variation $\frac{\sigma}{\mu}$ of the algorithms tested for 100 trial runs against the *random* dataset ($n = 10,000,000$). Data has been shuffled between runs.

The coefficient of variation $\frac{\sigma}{\mu}$ of run times of QuickselectAdaptive is one order of magnitude smaller than that of the baselines for medium and large data sets. The results for $C(n)$ and $S(n)$ indicate that QuickselectAdaptive makes a large reduction in the gap between theory and practice. Figure 10 and 11 reveal that the improvements also apply to real-world data. (Appendix A provides detailed numeric results.) Also, we speculate that further improvements will likely be difficult. $C(n)$ may still be improved significantly because $(2+\epsilon)n$ describes the worst, not average, case, but $S(n)$ is virtually at its theoretical optimum.

Acknowledgements. Thanks to Timon Gehr, Ivan Kazmenko, Scott Meyers, and Todd Millstein who reviewed drafts of this document. Teppo Niinimäki provided support code.

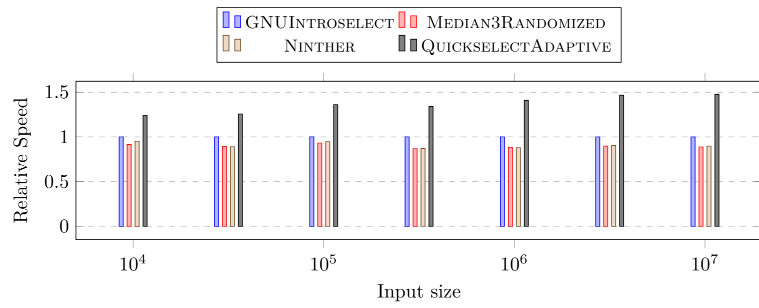


Figure 1 Speed relative to GNUIntroSelect (*random* dataset)

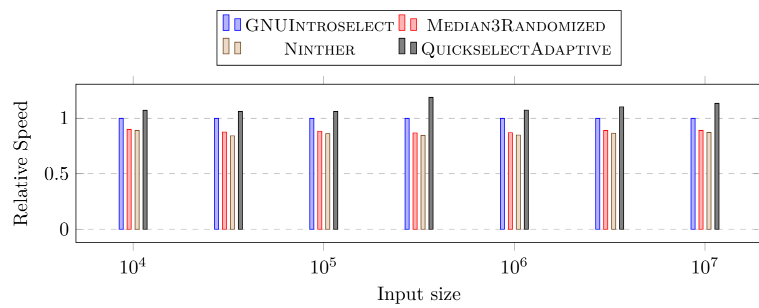


Figure 2 Speed relative to GNUIntroSelect (*random01* dataset).

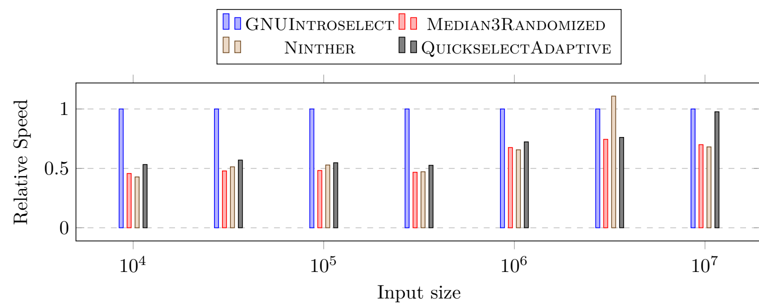


Figure 3 Speed relative to GNUIntroSelect (*m3killer* dataset).

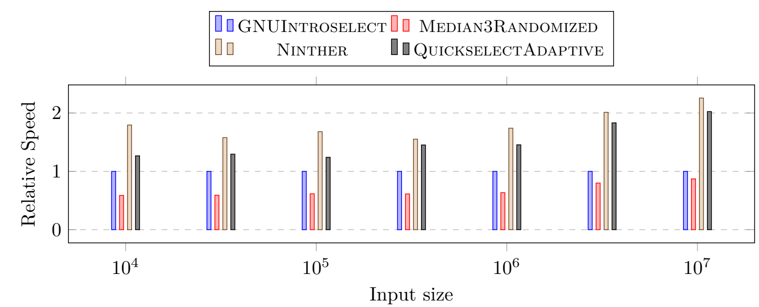


Figure 4 Speed relative to GNUIntroSelect (*organpipe* dataset).

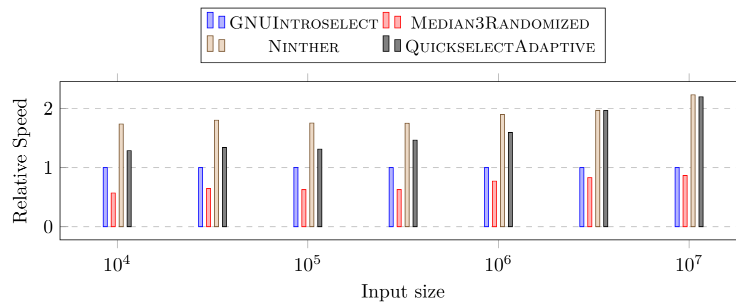


Figure 5 Speed relative to GNUIntroSelect (*sorted* dataset).

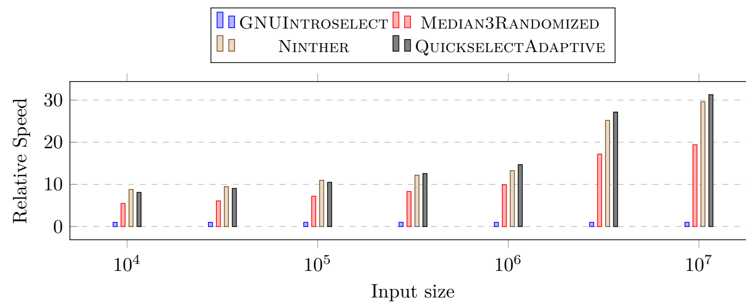


Figure 6 Speed relative to GNUIntroSelect (*rotated* dataset).

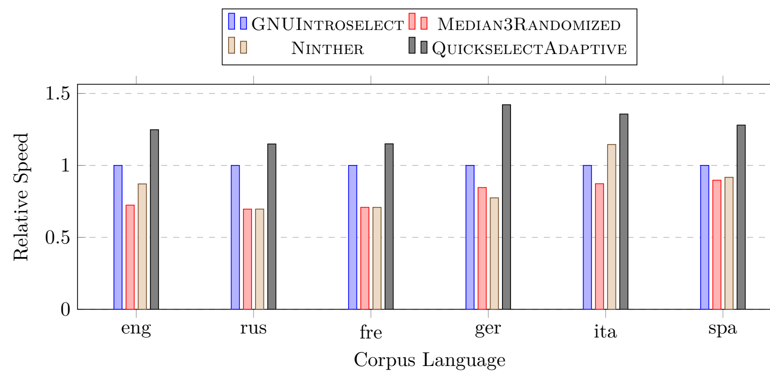


Figure 7 Speed relative to GNUIntroSelect (*googlebooks* dataset).

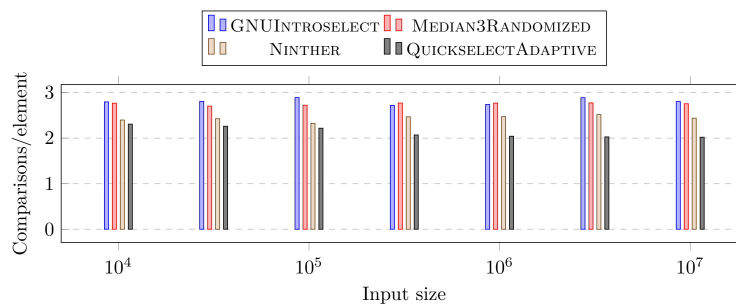


Figure 8 Comparisons per element (*random* dataset).

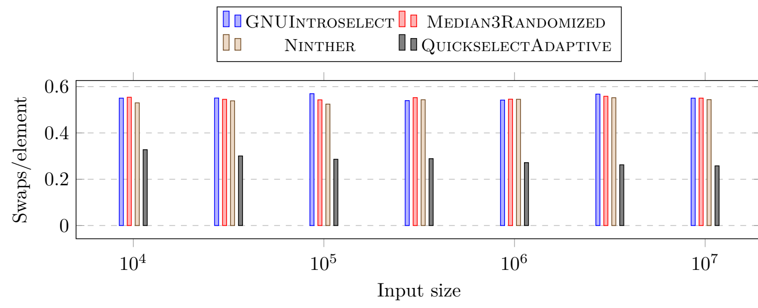


Figure 9 Swaps per element (*random* dataset).

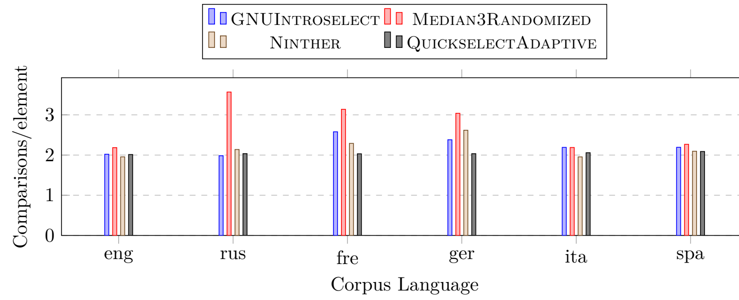


Figure 10 Comparisons per element (*googlebooks* dataset).

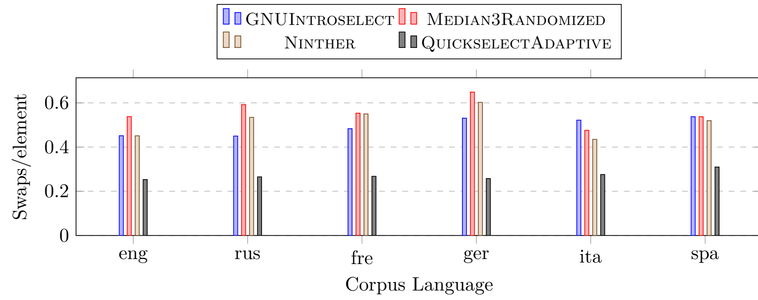


Figure 11 Swaps per element (*googlebooks* dataset).

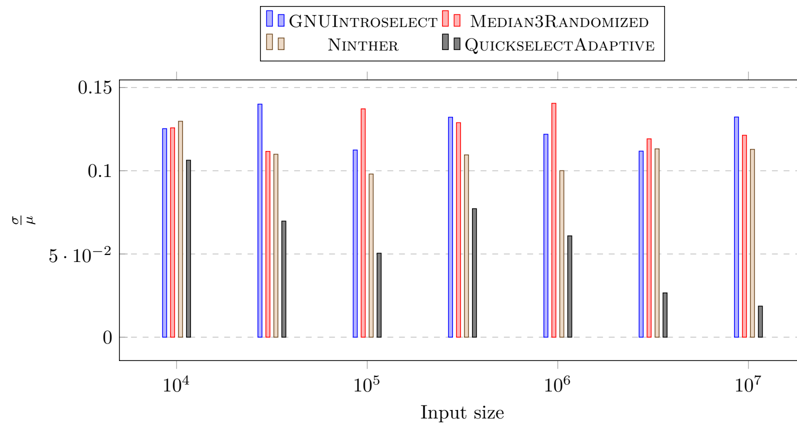


Figure 12 Coefficient of variation (*random* dataset).

References

1 Andrei Alexandrescu. Median of ninthers: code, data, and benchmarks. https://github.com/andralex/MedianOfNinthers, 2017.

2 Sebastiano Battiato, Domenico Cantone, Dario Catalano, Gianluca Cincotti, and Micha Hofri. An efficient algorithm for the approximate median selection problem. In *Algorithms and Complexity*, pages 226–238. Springer, 2000.

3 Jon L Bentley and M Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.

4 Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, August 1973. doi:10.1016/S0022-0000(73)80033-9.

5 Svante Carlsson and Mikael Sundström. *Algorithms and Computations: 6th International Symposium, ISAAC'95 Cairns, Australia, December 4–6, 1995 Proceedings*, chapter Linear-time in-place selection in less than 3n comparisons, pages 244–253. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. doi:10.1007/BFb0015429.

6 Ke Chen and Adrian Dumitrescu. Select with groups of 3 or 4. In *Algorithms and Data Structures: 14th International Symposium, WADS*, 2015.

7 Derrick Coetzee. An efficient implementation of Blum, Floyd, Pratt, Rivest, and Tarjan's worst-case linear selection algorithm, 2004. URL: http://moonflare.com/code/select/select.pdf.

8 Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

9 Jean Daligault and Conrado Martínez. On the variance of quickselect. In *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*, pages 205–210. Society for Industrial and Applied Mathematics, 2006.

10 Kevin Dinkel and Andrew Zizzi. Fast median finding on digital images. In *AIAA Regional Student Paper Conference*, 2012.

11 Dorit Dor and Uri Zwick. Median selection requires $(2+\epsilon)N$ comparisons. *SIAM J. Discret. Math.*, 14(3):312–325, March 2001. doi:10.1137/S0895480199353895.

12 Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Commun. ACM*, 18(3):165–172, March 1975. doi:10.1145/360680.360691.

13 GNU Team. Implementation of std::nth_element, 2016. [Online; accessed 27-Nov-2016]. URL: https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html.

14 Robin Griffin and K. A. Redish. Remark on algorithm 347 [m1]: An efficient algorithm for sorting with minimal storage. *Commun. ACM*, 13(1):54–, January 1970. doi:10.1145/361953.361993.

15 C. A. R. Hoare. Algorithm 63: Partition. *Commun. ACM*, 4(7):321–, July 1961. URL: http://doi.acm.org/10.1145/366622.366642, doi:10.1145/366622.366642.

16 C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961. doi:10.1145/366622.366644.

17 C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961. doi:10.1145/366622.366647.

18 Peter Kirschenhofer and Helmut Prodinger. Comparisons in Hoare's find algorithm. *Combinatorics, Probability and Computing*, 7:111–120, 3 1998. URL: http://journals.cambridge.org/article_S0963548397003325, doi:null.

19 Krzysztof C. Kiwiel. On Floyd and Rivest's SELECT Algorithm. *Theor. Comput. Sci.*, 347(1-2):214–238, November 2005. doi:10.1016/j.tcs.2005.06.032.

20 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

21 Tony W. Lai and Derick Wood. *SWAT 88: 1st Scandinavian Workshop on Algorithm Theory Halmstad, Sweden, July 5–8, 1988 Proceedings*, chapter Implicit selection, pages 14–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988. `doi:10.1007/3-540-19487-8_2`.

22 Conrado Martínez, Daniel Panario, and Alfredo Viola. *Mathematics and Computer Science II: Algorithms, Trees, Combinatorics and Probabilities*, chapter Analysis of Quickfind with Small Subfiles, pages 329–340. Birkhäuser Basel, Basel, 2002. `doi:10.1007/978-3-0348-8211-8_20`.

23 Conrado Martínez, Daniel Panario, and Alfredo Viola. Adaptive sampling strategies for quickselect. *ACM Trans. Algorithms*, 6(3):53:1–53:45, July 2010. `doi:10.1145/1798596.1798606`.

24 Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Joseph P. Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, et al. Quantitative analysis of culture using millions of digitized books. *science*, 331(6014):176–182, 2011.

25 David R. Musser. Introspective sorting and selection algorithms. *Software – Practice & Experience*, 27(8):983–993, 1997.

26 Himangi Saraogi. Median of medians algorithm, 2013. URL: `http://himangi774.blogspot.com/2013/09/median-of-medians.html`.

27 Uwe Schöning. Mastering the master theorem. *Bulletin of the EATCS*, 71:165–166, 2000.

28 SciPy.org. Implementation of `argpartition`, 2017. [Online; accessed Feb 9, 2017]. URL: `https://docs.scipy.org/doc/numpy/reference/generated/numpy.argpartition.html`.

29 R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 2011. URL: `https://books.google.com/books?id=idUdqdDXqnAC`.

30 Richard C. Singleton. Algorithm 347: An efficient algorithm for sorting with minimal storage [m1]. *Commun. ACM*, 12(3):185–186, March 1969. `doi:10.1145/362875.362901`.

31 J. W. Tukey. The ninther, a technique for low-effort robust (resistant) location in large samples. *Contributions to Survey Sampling and Applied Statistics in Honor of HO Hartley, Academic Press, New York*, pages 251–258, 1978.

32 Wikipedia. Median of medians, 2016. [Online; accessed 25-Feb-2016]. URL: `https://en.wikipedia.org/wiki/Median_of_medians`.

33 Andrew C. Yao and F. F. Yao. On the average-case complexity of selecting the k-th best. Technical report, Stanford University, Stanford, CA, USA, 1979.

34 Chee K. Yap. New upper bounds for selection. *Commun. ACM*, 19(9):501–508, September 1976. `doi:10.1145/360336.360339`.

A Additional Measurement Results

Table 1 Run times in milliseconds (*random* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
10,000	$7.45 \cdot 10^{-2}$	0.19	$8.14 \cdot 10^{-2}$	$7.81 \cdot 10^{-2}$	$6.01 \cdot 10^{-2}$
31,620	0.22	0.61	0.25	0.25	0.18
100,000	0.72	1.97	0.77	0.76	0.53
316,220	2.15	6.19	2.48	2.47	1.61
1,000,000	6.99	19.74	7.90	7.96	4.96
3,162,280	23.11	62.86	25.73	25.52	15.75
10,000,000	71.67	198.41	80.74	79.84	48.57

Table 2 Run times in milliseconds (*random01* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$5.76\cdot10^{-2}$	0.14	$6.40\cdot10^{-2}$	$6.46\cdot10^{-2}$	$5.37\cdot10^{-2}$
$31,620$	0.17	0.44	0.20	0.21	0.16
$100,000$	0.55	1.35	0.62	0.64	0.52
$316,220$	1.71	4.19	1.97	2.02	1.44
$1,000,000$	5.50	13.31	6.33	6.47	5.12
$3,162,280$	17.92	43.54	20.13	20.70	16.26
$10,000,000$	57.47	142.78	64.48	65.96	50.66

Table 3 Run times in milliseconds (*m3killer* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$8.58\cdot10^{-3}$	0.11	$1.88\cdot10^{-2}$	$2.01\cdot10^{-2}$	$1.61\cdot10^{-2}$
$31,620$	$2.79\cdot10^{-2}$	0.28	$5.83\cdot10^{-2}$	$5.43\cdot10^{-2}$	$4.89\cdot10^{-2}$
$100,000$	$8.65\cdot10^{-2}$	1.05	0.18	0.16	0.16
$316,220$	0.25	2.42	0.53	0.53	0.47
$1,000,000$	1.25	9.43	1.85	1.91	1.73
$3,162,280$	5.42	33.15	7.28	4.89	7.13
$10,000,000$	19.06	109.99	27.24	27.99	19.52

Table 4 Run times in milliseconds (*organpipe* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$5.19\cdot10^{-3}$	$2.24\cdot10^{-2}$	$8.83\cdot10^{-3}$	$2.89\cdot10^{-3}$	$4.10\cdot10^{-3}$
$31,620$	$1.60\cdot10^{-2}$	$6.24\cdot10^{-2}$	$2.71\cdot10^{-2}$	$1.01\cdot10^{-2}$	$1.24\cdot10^{-2}$
$100,000$	$4.86\cdot10^{-2}$	0.29	$7.90\cdot10^{-2}$	$2.90\cdot10^{-2}$	$3.92\cdot10^{-2}$
$316,220$	0.15	0.74	0.24	$9.60\cdot10^{-2}$	0.10
$1,000,000$	0.47	2.61	0.74	0.27	0.32
$3,162,280$	2.33	10.00	2.92	1.16	1.27
$10,000,000$	8.37	37.28	9.61	3.71	4.13

Table 5 Run times in milliseconds (*sorted* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$9.92\cdot10^{-3}$	$5.54\cdot10^{-2}$	$1.74\cdot10^{-2}$	$5.70\cdot10^{-3}$	$7.71\cdot10^{-3}$
$31,620$	$3.12\cdot10^{-2}$	0.14	$4.82\cdot10^{-2}$	$1.73\cdot10^{-2}$	$2.32\cdot10^{-2}$
$100,000$	$9.51\cdot10^{-2}$	0.79	0.15	$5.41\cdot10^{-2}$	$7.22\cdot10^{-2}$
$316,220$	0.30	1.60	0.47	0.17	0.20
$1,000,000$	1.40	5.84	1.81	0.74	0.88
$3,162,280$	4.93	20.28	5.94	2.50	2.51
$10,000,000$	17.10	81.60	19.67	7.66	7.77

Table 6 Run times in milliseconds (*rotated* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$9.45{\cdot}10^{-2}$	$5.29{\cdot}10^{-2}$	$1.73{\cdot}10^{-2}$	$1.08{\cdot}10^{-2}$	$1.17{\cdot}10^{-2}$
$31,620$	0.32	0.15	$5.18{\cdot}10^{-2}$	$3.33{\cdot}10^{-2}$	$3.48{\cdot}10^{-2}$
$100,000$	1.12	0.59	0.16	0.10	0.11
$316,220$	3.92	1.57	0.47	0.32	0.31
$1,000,000$	16.91	5.90	1.70	1.27	1.15
$3,162,280$	98.91	20.29	5.75	3.93	3.64
$10,000,000$	366.41	82.09	18.89	12.38	11.71

Table 7 Run times in milliseconds (*Google Ngram* dataset).

Corpus	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
eng	117.81	396.29	162.80	135.19	94.40
fre	49.69	169.70	70.12	70.14	43.21
ger	77.98	222.23	92.16	100.67	54.86
ita	37.96	106.29	43.49	33.15	27.98
rus	64.83	224.04	93.15	93.12	56.44
spa	50.01	134.70	55.74	54.52	39.07

Table 8 Comparisons per element (*random* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	2.80	6.75	2.77	2.40	2.31
$31,620$	2.81	6.91	2.70	2.43	2.26
$100,000$	2.89	7.14	2.72	2.32	2.22
$316,220$	2.72	7.21	2.77	2.47	2.07
$1,000,000$	2.74	7.30	2.77	2.47	2.04
$3,162,280$	2.89	7.33	2.77	2.52	2.03
$10,000,000$	2.80	7.34	2.75	2.44	2.02

Table 9 Comparisons per element (*googlebooks* dataset).

Corpus	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
eng	2.02	7.29	2.18	1.95	2.01
fre	2.58	7.45	3.14	2.29	2.03
ger	2.38	7.54	3.04	2.62	2.03
ita	2.19	7.58	2.19	1.95	2.06
rus	1.98	7.44	3.57	2.14	2.03
spa	2.19	7.38	2.27	2.09	2.09

Table 10 Swaps per element (*random* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	0.55	3.28	0.55	0.53	0.33
$31,620$	0.55	3.35	0.55	0.54	0.30
$100,000$	0.57	3.46	0.54	0.52	0.29
$316,220$	0.54	3.50	0.55	0.54	0.29
$1,000,000$	0.54	3.54	0.55	0.55	0.27
$3,162,280$	0.57	3.54	0.56	0.55	0.26
$10,000,000$	0.55	3.55	0.55	0.54	0.26

Table 11 Swaps per element (*googlebooks* dataset).

Corpus	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
eng	0.45	3.43	0.54	0.45	0.25
fre	0.48	3.50	0.55	0.55	0.27
ger	0.53	3.49	0.65	0.60	0.26
ita	0.52	3.55	0.48	0.43	0.28
rus	0.45	3.50	0.59	0.53	0.27
spa	0.54	3.44	0.54	0.52	0.31

Table 12 Coefficient of variation of run time (*random* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	0.13	0.11	0.13	0.13	0.11
$31,620$	0.14	$8.63{\cdot}10^{-2}$	0.11	0.11	$6.97{\cdot}10^{-2}$
$100,000$	0.11	$2.57{\cdot}10^{-2}$	0.14	$9.80{\cdot}10^{-2}$	$5.05{\cdot}10^{-2}$
$316,220$	0.13	$5.13{\cdot}10^{-2}$	0.13	0.11	$7.72{\cdot}10^{-2}$
$1,000,000$	0.12	$1.56{\cdot}10^{-2}$	0.14	0.10	$6.09{\cdot}10^{-2}$
$3,162,280$	0.11	$1.61{\cdot}10^{-2}$	0.12	0.11	$2.66{\cdot}10^{-2}$
$10,000,000$	0.13	$1.57{\cdot}10^{-2}$	0.12	0.11	$1.87{\cdot}10^{-2}$

Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*

Antoine Limasset[1], Guillaume Rizk[2], Rayan Chikhi[3], and Pierre Peterlongo[4]

1 **IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France**
2 **IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France**
3 **CNRS, CRIStAL, Université de Lille, Inria Lille – Nord Europe, Lille, France**
4 **IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France**

Abstract

Minimal perfect hash functions provide space-efficient and collision-free hashing on static sets. Existing algorithms and implementations that build such functions have practical limitations on the number of input elements they can process, due to high construction time, RAM or external memory usage. We revisit a simple algorithm and show that it is highly competitive with the state of the art, especially in terms of construction time and memory usage. We provide a parallel C++ implementation called *BBhash*. It is capable of creating a minimal perfect hash function of 10^{10} elements in less than 7 minutes using 8 threads and 5 GB of memory, and the resulting function uses 3.7 bits/element. To the best of our knowledge, this is also the first implementation that has been successfully tested on an input of cardinality 10^{12}. Source code: `https://github.com/rizkg/BBHash`.

1998 ACM Subject Classification H.3.1 Content Analysis and Indexing, E.2 Data Storage Representations

Keywords and phrases Minimal Perfect Hash Functions, Algorithms, Data Structures, Big Data

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.25

1 Introduction

Given a set S of N elements (*keys*), a minimal perfect hash function (`MPHF`) is an injective function that maps each key of S to an integer in the interval $[1, N]$. In other words, an `MPHF` labels each key of S with integers in a collision-free manner, using the smallest possible integer range. A remarkable property is the small space in which these functions can be stored: only a couple of bits per key, independently of the size of the keys. Furthermore, an `MPHF` query is done in constant time. While an `MPHF` could be easily obtained using a key-value store (e.g. a hash table), such a representation would occupy an unreasonable amount of space, with both the keys and the integer labels stored explicitly.

The theoretical minimum amount of space needed to represent an `MPHF` is known to be $\log_2(e)N \approx 1.44N$ bits [10, 14]. In practice, for large key sets (billions of keys), many implementations achieve less than $3N$ bits per key, regardless of the number of keys [2, 9]. However no implementation comes asymptotically close to the lower bound for large key sets. Given that `MPHF`s are typically used to index huge sets of strings, e.g. in bioinformatics [6, 7, 8], in network applications [12], or in databases [5], lowering the representation space is of interest.

* This work was funded by French ANR-12-BS02-0008 Colib'read project.

© Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo;
licensed under Creative Commons License CC-BY

16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 25; pp. 25:1–25:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We observe that in many of these applications, MPHFs are actually used to construct static dictionaries, i.e. key-value stores where the set of keys is fixed and never updated [6, 8]. Assuming that the user only queries the MPHF to get values corresponding to keys that are guaranteed to be in the static set, the keys themselves do not necessarily need to be stored in memory. However the associated values in the dictionary typically do need to be stored, and they often dwarf the size of the MPHF. The representation of such dictionaries then consists of two components: a space-efficient MPHF, and a relatively more space-expensive set of values. In such applications, whether the MPHF occupies 1.44 bits or 3 bits per key is thus arguably not a critical aspect.

In practice, a significant bottleneck for large-scale applications is the construction step of MPHFs, both in terms of memory usage and computation time. Constructing MPHFs efficiently is an active area of research. Many recent MPHF construction algorithms are based on efficient peeling of hypergraphs [1, 3, 4, 11]. However, they require an order of magnitude more space during construction than for the resulting data structure. For billions of keys, while the MPHF itself can easily fit in main memory of a commodity computer, its construction algorithm requires large-memory servers. To address this, Botelho and colleagues [4] propose to divide the problem by building many smaller MPHFs, while Belazzougui *et al.* [1] propose an external-memory algorithm for hypergraph peeling. Very recently, Genuzio *et al.* [11] demonstrated practical improvements to the Gaussian elimination technique, that make it competitive with [1] in terms of construction time, lookup time and space of the final structure. These techniques are, to the best of our knowledge, the most scalable solutions available. However, when evaluating existing implementations, the construction of MPHFs for sets that significantly exceed a billion keys remains prohibitive in terms of time and space usage.

A simple idea has been explored by previous works [6, 12, 16] for constructing PHFs (Perfect Hash Functions, non minimal) or MPHFs using arrays of bits, or fingerprints. However, it has received relatively less attention compared to other hypergraph-based methods, and no implementation is publicly available in a stand-alone MPHF library. In this article we revisit this idea, and introduce novel contributions: a careful analysis of space usage during construction, and an efficient, parallel implementation along with an extensive evaluation with respect to the state of the art. We show that it is possible to construct an MPHF using almost as little memory as the space required by the final structure, without partitioning the input. We propose a novel implementation called *BBhash* ("Basic Binary representAtion of Successive Hashing") with the following features:

- construction space overhead is small compared to the space occupied by the MPHF,
- multi-threaded,
- scales up to to very large key sets (tested with up to 1 trillion keys).

To the best of our knowledge, there does not exist another usable implementation that satisfies any two of the features above. Furthermore, the algorithm enables a time/memory trade-off: faster construction and faster query times can be obtained at the expense of a few more bits per element in the final structure and during construction. We created an MPHF for ten billion keys in 6 minutes 47 seconds and less than 5 GB of working memory, and an MPHF for a trillion keys in less than 36 hours and 637 GB memory. Overall, with respect to other available MPHF construction approaches, our implementation is at least two orders of magnitudes more space-efficient when considering internal and external memory usage during construction, and at least one order of magnitude faster. The resulting MPHF has slightly higher space usage and faster or comparable query times than other methods.

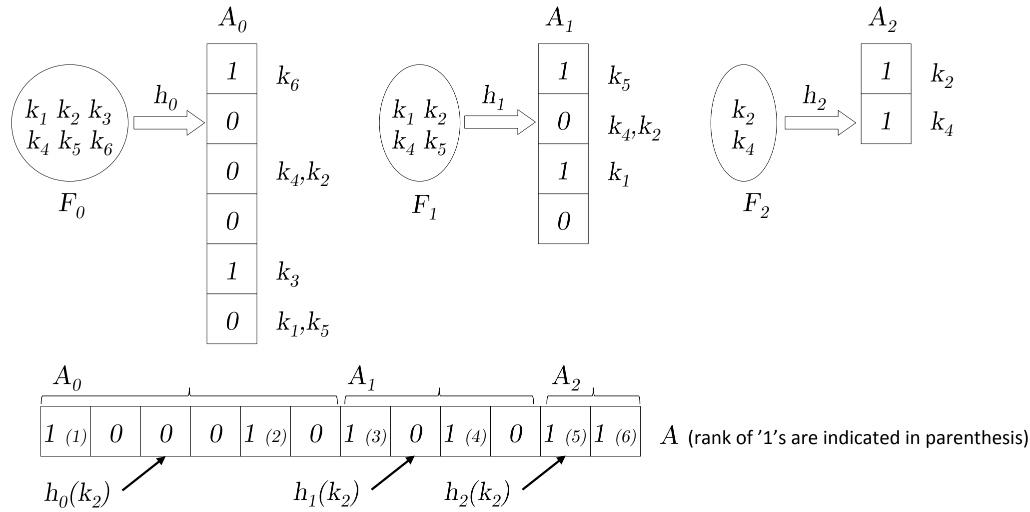


Figure 1 MPHF construction and query example. The input is a set F_0 composed of $N = 6$ keys (k_1 to k_6). All keys are hashed using a hash function h_0 and are attempted to be placed in an array A_0 at positions given by the hash function. The keys k_3 and k_6 do not have collisions in the array, thus the corresponding bits in A_0 are set to '1'. The other keys from F_0 that are involved in collisions are placed in a new set F_1. In the second level, keys from F_1 are hashed using a hash function h_1. Keys k_1 and k_5 are uniquely placed while k_2 and k_4 collide, thus they are then stored in the set F_2. With the hash function h_2, the keys from F_2 have no collision, and the process finishes. The MPHF query operation is very similar to the construction algorithm. Let A be the the concatenation of A_0, A_1, A_2 (see bottom part of the figure). To query k_2, the key is first hashed with h_0. The associated value in A_0 is '0', so k_2 is then hashed with h_1. The value associated in A_1 is again '0'. When finally hashed with h_2, the value associated in A_2 is '1' and thus the query stops here. The index returned by the MPHF is the rank of this '1' (here, 5) in A. In this example, the MPHF values returned when querying k_1, k_2, k_3, k_4, k_5 and k_6 are respectively 4,5,2,6,3, and 1.

2 Efficient construction of minimal perfect hash function

2.1 Method overview

Our MPHF construction procedure revisits previously published techniques [6, 12]. Given a set F_0 of keys, a classical hash function h_0 maps keys to an integer in $[1, |F_0|]$. A bit array A_0 of size $|F_0|$ is created such that there is a 1 at position i if and only if exactly one element of F_0 has a hash value of i. We say that there is a *collision* whenever two keys in F_0 have the same hash value. Keys from F_0 that were involved in a collision are inserted into a new set F_1. The process repeats with F_1 and a new hash function h_1. A new bit array A_1 of size $|F_1|$ is created using the same procedure as for A_0 (except that F_1 is used instead of F_0, and h_1 instead of h_0). The process is repeated with $F_2, F_3, \ldots$ until one of these sets, $F_{\text{last}+1}$, is empty.

We obtain an MPHF by concatenating the bit arrays $A_0, A_1, \ldots, A_{\text{last}}$ into an array A. To perform a query, a key is hashed successively with hash functions $h_0, h_1, \ldots$ as long as the value in A_i ($i \geq 0$) at the position given by the hash function h_i is 0. Eventually, by construction, we reach a 1 at some position of A for some $i = d$. We say that the *level* of the key is d. The index returned by the MPHF is the rank of this one in A. See Figure 1 for an example.

2.2 Algorithm details

2.2.1 Collision detection

During construction at each level d, collisions are detected using a temporary bit array C_d of size $|A_d|$. Initially all C_d bits are set to '0'. A bit of $C_d[i]$ is set to '1' if two or more keys from F_d have the same value i given by hash function h_d. Finally, if $C_d[i] = 1$, then $A_d[i] = 0$. Formally:

$$C_d[i] = 1 \Rightarrow A_d[i] = 0;$$
$$(h_d[x] = i \text{ and } A_d[i] = 0 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 1\,(\text{and } C_d[i] = 0)\,;$$
$$(h_d[x] = i \text{ and } A_d[i] = 1 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 0 \text{ and } C_d[i] = 1.$$

2.2.2 Queries

A query of a key x is performed by finding the smallest d such that $A_d[h_d(x)] = 1$. The (non minimal) hash value of x is then $(\sum_{i<d} |F_i|) + h_d(x)$.

2.2.3 Minimality

To ensure that the image range of the function is $[1, |F_0|]$, we compute the cumulative rank of each '1' in the bit arrays A_i. Suppose, that d is the smallest value such that $A_d[h_d(x)] = 1$. The minimal perfect hash value is given by $\sum_{i<d}(weight(A_i) + rank(A_d[h_d(x)])$, where $weight(A_i)$ is the number of bits set to '1' in the A_i array, and $rank(A_d[y])$ is the number of bits set to 1 in A_d within the interval $[0, y]$, thus $rank(A_d[y]) = \sum_{j<y} A_d[j]$. This is a classic method also used in other MPHFs [3].

2.2.4 Faster query and construction times (parameter γ)

The running time of the construction depends on the number of collisions on the A_d arrays, at each level d. One way to reduce the number of collisions, hence to place more keys at each level, is to use bit arrays (A_d and C_d) larger than $|F_d|$. We introduce a parameter $\gamma \in \mathbb{R}$, $\gamma \geq 1$, such that $|C_d| = |A_d| = \gamma|F_d|$. With $\gamma = 1$, the size of A is minimal. With $\gamma \geq 2$, the number of collisions is significantly decreased and thus construction and query times are reduced, at the cost of a larger `MPHF` structure size. The influence of γ is discussed in more detail in the following analyses and results.

2.3 Analysis

Proofs of the following observations and lemma are given in the Appendix.

2.3.1 Size of the `MPHF`

The expected size of the structure can be determined using a simple argument, previously made in [6]. When $\gamma = 1$, the expected number of keys which do not collide at level d is $|A_d|e^{-1}$, thus $|A_d| = |A_{d-1}|(1 - e^{-1}) = |A_0|(1 - e^{-1})^d$. In total, the expected number of bits required by the hashing scheme is $\sum_{d\geq 0} |A_d| = N \sum_{d\geq 0}(1 - e^{-1})^d = eN$, with N being the total number of input keys ($N = |F_0|$). Note that consequently the image of the hash function is also in $[1, eN]$, before minimization using the rank technique. When $\gamma \geq 1$, the expected proportion of keys without collisions at each level d is $|A_d|e^{-\frac{1}{\gamma}}$. Since each A_d no

longer uses one bit per key but γ bits per key, the expected total number of bits required by the MPHF is $\gamma e^{\frac{1}{\gamma}} N$.

2.3.2 Space usage during construction

We analyze the disk space used during construction. Recall that during construction of level d, a bit array C_d of size $|A_d|$ is used to record collisions. Note that the C_d array is only needed during the d-th level. It is deleted before level $d+1$. The total memory required during level d is $\sum_{i \leq d}(|A_i|) + |C_d| = \sum_{i<d}(|A_i|) + 2|A_d|$.

▶ **Lemma 1.** *For $\gamma > 0$, the space of our MPHF is $S = \gamma e^{\frac{1}{\gamma}} N$ bits. The maximal space during construction is S when $\gamma \leq \log(2)^{-1}$, and $2S$ bits otherwise.*

A full proof of the Lemma is provided in the Appendix.

3 Implementation

We present *BBhash*, a C++ implementation available at `http://github.com/rizkg/BBHash`. We describe in this section some design key choices and optimizations.

3.1 Rank structure

We use a classical technique to implement the rank operation: the ranks of a fraction of the '1's present in A are recorded, and the ranks in-between are computed dynamically using the recorded ranks as checkpoints.

In practice 64 bit integers are used for counters, which is enough for realistic use of an MPHF, and placed every 512 positions by default. These values were chosen as they offer a good speed/memory trade-off, increasing the size of the MPHF by a factor 1.125 while achieving good query performance. The total size of the MPHF is thus $(1 + \frac{64}{512})\gamma e^{\frac{1}{\gamma}} N$.

3.2 Parallelization

Parallelization is achieved by partitioning keys over several threads. The algorithm presented in Section 2 is executed on multiple threads concurrently, over the same memory space. Built-in compiler functions (e.g. *sync_fetch_and_or*) are used for concurrent access in the A_i arrays. The efficiency of this parallelization scheme is shown in the Results section, but note that it is fundamentally limited by random memory accesses to the A_i arrays which incur cache misses.

3.3 Hash functions

The MPHF construction requires classical hash functions. Other authors have observed that common hash functions behave practically as well as fully random hash functions [2]. We therefore choose to use xor-shift based hash functions [13] for their efficiency both in terms of computation speed and distribution uniformity [15].

3.4 Disk usage

In the applications we consider, key sets are typically too big to fit in RAM. Thus we propose to read them on the fly from disk. There are mainly two distinct strategies regarding the disk usage during construction: 1/ during each level d, keys that are to be inserted in the set

F_{d+1} are written directly to disk. The set F_{d+1} is then read during level $d+1$ and erased before level $d+2$; or 2/ at each level all keys from the original input key file are read and queried in order to determine which keys were already assigned to a level $i < d$, and which would belong to F_d. When the key set becomes small enough (below user-defined threshold) it is loaded in ram to avoid costly re-computation from scratch at each level.

The first strategy obviously provides faster construction at the cost of temporary disk usage. At each level $d > 0$, two temporary key files are stored on disk: F_d and F_{d+1}. The highest disk usage is thus achieved during level 1, i.e. by storing $|F_1|+|F_2| = |F_0|((1-e^{-1/\gamma})+(1-e^{-1/\gamma})^2)$ elements. With $\gamma = 1$, this represents $\approx 1.03N$ elements, thus the construction overhead on disk is approximately the size of the input key file. Note that with $\gamma = 2$ (resp. $\gamma = 5$), this overhead diminishes and becomes a ratio of ≈ 0.55 (resp. ≈ 0.21) the size of the input key file.

The first strategy is the default strategy proposed in our implementation. The second one has also been implemented and can be optionally switched on.

3.5 Termination

The expected number of unplaced keys decreases exponentially with the number of levels but is not theoretically guaranteed to reach zero in a finite number of steps. To ensure termination of the construction algorithm, in our implementation a maximal number D of levels is fixed. Then, the remaining keys are inserted into a regular hash table. Value D is a parameter, its default value is $D = 25$ for which the expected number of keys stored in this hash table is $\approx 10^{-5}N$ for $\gamma = 1$ and becomes in practice negligible for $\gamma \geq 2$, allowing the size overhead of the final hash table to be negligible regarding the final `MPHF` size.

4 Results

We evaluated the performance *BBhash* for the construction of large `MPHF`s. We generated files containing various numbers of keys (from 1 million to 1 trillion keys). In our tests, a key is a binary representation of a pseudo-random positive integer in $[0; 2^{64}]$. Within each file, each key is unique. We also performed a test where input keys are strings (n-grams) to ensure that using integers as keys does not bias results. Tests were performed on a cluster node with a Intel© Xeon© CPU E5-2660 v3 2.60GH 20-core CPU, 256 GB of memory, and a mechanical hard drive. Except for the experiment with 10^{12} keys, running times include the time needed to read input keys from disk. Note that files containing key sets may be cached in memory by the operating system, and all evaluated methods benefit from this effect during `MPHF` construction. We refer to the Appendix for the specific commands and parameters used in these experiments.

We first analyzed the influence of the γ value (the main parameter of *BBhash*), then the effect of using multiple threads depending on the parallelization strategy. Second, we compared *BBhash* with other state-of-the-art methods. Finally, we performed an `MPHF` construction on 10^{12} elements.

4.1 Influence of the γ parameter

We report in Figure 2 (left) the construction times and the mean query times, as well as the size of the produced `MPHF`, with respect to several γ values. The main observation is that $\gamma \geq 2$ drastically accelerates construction and query times. This is expected since large γ values allow more elements to be placed in the first levels of the `MPHF`; thus limiting the

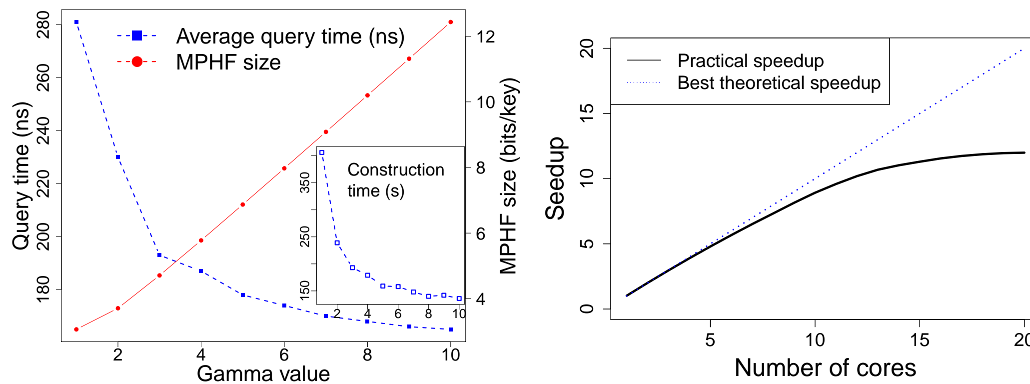


Figure 2 Left: Effects of the gamma parameter on the performance of *BBhash* when run on a set composed of one billion keys, when executed on a single CPU thread. Times and MPHF size behave accordingly to the theoretical analysis, respectively $O(e^{(1/\gamma)})$, and $O(\gamma e^{(1/\gamma)})$. Right: Performance of the *BBhash* construction time according to the number of cores, using $\gamma = 2$.

number of times each key is hashed to compute its level. In particular, for keys placed in the very first level, the query time is limited to a single hashing and a memory access. The average level of all keys is $e^{(1/\gamma)}$, we therefore expect construction and query times to decrease when γ increases. However, larger γ values also incur larger MPHF sizes. One observes that $\gamma > 5$ values seem to bring very little advantage at the price of higher space requirements. A related work used $\gamma = 1$ in order to minimize the MPHF size [6]. Here, we argue that using γ values larger than 1 has significant practical merits. In our tests, we often used $\gamma = 2$ as it yields an attractive time/space trade-off during construction and queries.

4.2 Parallelization performance

We evaluated the capability of our implementation to make use of multiple CPU cores. In Figure 2 (right), we report the construction times with respect to the number of threads. We observe a near-ideal speed-up with respect to the number of threads with diminishing returns when using more than 10 threads, which is likely due to cache misses that induce a memory access bottleneck.

In addition to these results, we applied *BBhash* on a key set of 10 billion keys and on a key set of 100 billion keys, again using default parameters and 8 threads. The memory usage was respectively 4.96GB and 49.49GB, and the construction time was respectively 462 seconds and 8913 seconds, showing the scalability of *BBhash*.

4.3 Comparisons with state of the art methods

We compared *BBhash* with state-of-the-art MPHF methods. CHD (http://cmph.sourceforge.net/) is an implementations of the compressed hash-and-displace algorithm [2]. EMPHF [1] is based on random hypergraph peeling, and the HEM [4] implementation in EMPHF is based on partitioning the input data. Sux4J is a Java implementation of [11]. We did not include other methods cited earlier because they do not provide an implementation [12, 16] or the software integrates a non-minimal perfect hash function that is not stand-alone [6]. However single-threaded results presented in [16] show that construction times and MPHF sizes are comparable to ours, query times are significantly longer, and no

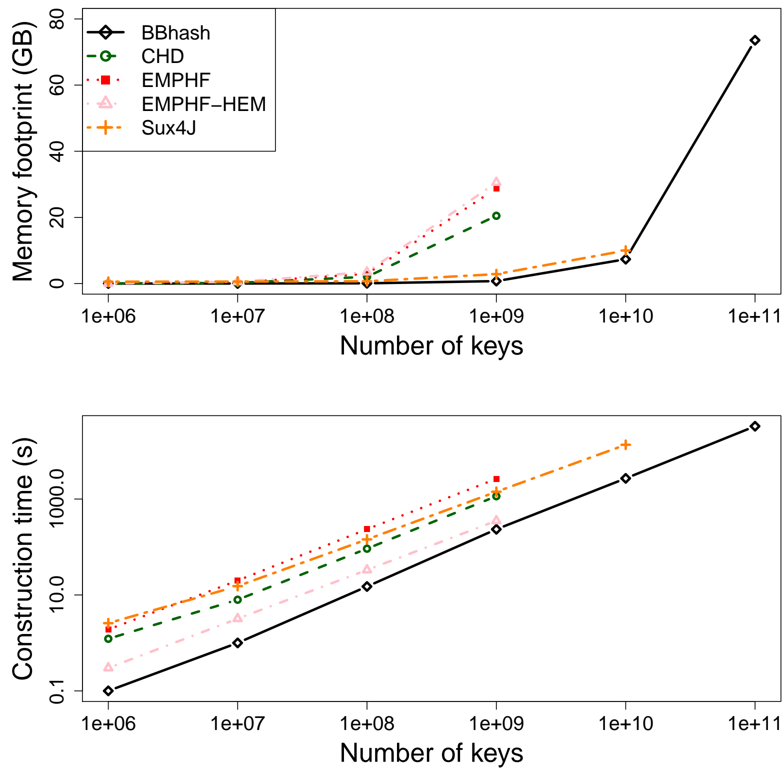


Figure 3 Memory footprint and construction time with respect to the number of keys. All libraries were run using default parameters, including $\gamma = 2$ for *BBhash*. For a fair comparison, *BBhash* was executed on a single CPU thread. Except for Sux4J, missing data points correspond to runs that exceeded the amount of available RAM. Sux4J limit comes from the disk usage, estimated at approximately 4TB for 10^{11} keys.

indication is provided about the memory usage during construction. Our benchmark code is available at `https://github.com/rchikhi/benchmphf`.

Figure 3 shows that all evaluated methods are able to construct `MPHF`s that contain a billion elements, but only *BBhash* scales up to datasets that contain 10^{11} elements and more. Overall, *BBhash* shows consistently better time and memory usage during construction.

We additionally compared the resulting `MPHF` size, i.e. the space of the data structure returned by the construction algorithm, and the mean query time across all libraries on a dataset consisting of a billion keys (Table 1). `MPHF`s produced by *BBhash* range from 2.89 bits/key (when $\gamma = 1$ and ranks are sampled every 1024 positions) to 6.9 bits/key (when $\gamma = 5$ and a rank sampling of 512). The 0-0.8 bits/key size difference between our implementation and the theoretical space usage of the *BBhash* structure size is due to additional space used by the rank structure. We believe that a reasonable compromise in terms of query time and structure size is 3.7 bits/key with $\gamma = 2$ and a rank sampling of 512, which is marginally larger than the `MPHF` sizes of other libraries (ranging from 2.6 to 3.5 bits/key). As we argued in the Introduction, using one more bit per key seems to be a reasonable trade-off for performance.

Construction times vary by one or two orders of magnitude across methods, *BBhash* being the fastest. With default parameters ($\gamma = 2$, rank sampling of 512), *BBhash* has a

Table 1 Performance of different MPHF algorithms applied on a key set composed of 10^9 64-bits random integers, of size 8GB. Each time result is the average value over three tests. The 'nodisk' row implements the second strategy described in Section 3.4, and the 'minirank' row samples ranks every 1024 positions instead of 512 by default. *The column "*Const. time*" indicates the construction time in seconds. In the case of *BBhash*, the first value is the construction time using eight CPU threads and the second value in parenthesis is the one using one CPU thread. **The column "*Const. memory*" indicates the RAM used during the MPHF construction, in bits/key and the total in MB in parenthesis. † The memory usages of EMPHF and EMPHF HEM reflect the use of memory-mapped files (mmap scheme).

Method	Query time (ns)	MPHF size (bits/key)	Const. time* (s)	Const. memory**	Disk. usage (GB)
BBhash $\gamma = 1$	271	3.1	60 (393)	3.2 (376)	8.23
BBhash $\gamma = 1$ minirank	279	2.9	61(401)	3.2 (376)	8.23
BBhash $\gamma = 2$	216	3.7	35 (229)	4.3 (516)	4.45
BBhash $\gamma = 2$ nodisk	216	3.7	80 (549)	6.2 (743)	0
BBhash $\gamma = 5$	179	6.9	25 (162)	10.7 (1,276)	1.52
EMPHF	246	2.9	2,642	247.1 (29,461)†	20.8
EMPHF HEM	581	3.5	489	258.4 (30,798)†	22.5
CHD	1037	2.6	1,146	176.0 (20,982)	0
Sux4J	252	3.3	1,418	18.10 (2,158)	40.1

construction memory footprint 40× to 60× smaller than other libraries except for Sux4j, for which *BBhash* remains 4× smaller. Query times are roughly within an order of magnitude ($179 - 1037$ ns) of each other across methods, with a slight advantage for *BBhash* when $\gamma \geq 2$. Sux4j achieves an attractive balance with low construction memory and query times, but high disk usage. In our tests, the high disk usage of Sux4j was a limiting factor for the construction of very large MPHFs.

Note that EMPHF, EMPHF HEM and Sux4j implement a disk partitioning strategy, that could in principle also be applied to others methods, including ours. Instead of creating a single large MPHF, they partition the set of input keys on disk and construct many small MPHFs independently. In theory this technique allows to engineer the MPHF construction algorithm to use parallelism and lower memory, at the expense of higher disk usage. In practice we observe that the existing implementations that use this technique are not parallelized. While EMPHF and EMPHF HEM used relatively high memory in our tests (around 30 GB for 1 billion elements) due to memory-mapped files, they also completed the construction successfully on another machine that had 16 GB of available memory. However, we observed what appears to be limitations in the scalability of the scheme: we were unable to run EMPHF and EMPHF HEM on an input of 10 billion elements using 256 GB of memory. Regardless, we view this partitioning technique as promising but orthogonal to the design of efficient "monolithic" MPHFs constructions such as *BBhash*.

4.4 Performance on an actual dataset

In order to ensure that using pseudo-random integers as keys does not bias results, we ran *BBhash* using strings as keys. We used n-grams extracted from the Google Books Ngram dataset[1], version 20120701. On average the n-gram size is 18. We also generated random

[1] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

Table 2 Performance of *BBhash* ($\gamma = 2$, 8 threads) when using ASCII strings as keys.

Dataset	Query time (ns)	MPHF size (bits/key)	Const. time (s)
10^8 Random strings	325	3.7	35
10^8 Ngrams	296	3.7	37

words of size 18. As reported in Table 2, we obtained highly similar results to those obtained with random integer keys.

4.5 Indexing a trillion keys

We performed a very large-scale test by creating an MPHF for 10^{12} keys. For this experiment, we used a machine with 750 GB of RAM. Since storing that many keys would require 8 TB of disk space, we instead used a procedure that deterministically generates a stream of 10^{12} pseudo-random integers in $[0, 2^{64} - 1]$. We considered the streamed values as input keys without writing them to disk. In addition, key sets of cardinality below 20 billion (2% of the input) were stored in memory to avoid re-computation from scratch at each subsequent level. Thus, the reported computation time should not be compared to previously presented results as this experiment has no disk accesses. The test was performed using $\gamma = 2$, 24 threads.

Creating the MPHF took 35.4 hours and required 637 GB RAM. This memory footprint is roughly separated between the bit arrays ($\approx$ 459 GB) and the memory required for loading 20 billion keys in memory ($\approx$ 178 GB). The final MPHF occupied 3.71 bits per key.

5 Conclusion

We have proposed a resource-efficient and highly scalable algorithm for constructing and querying MPHFs. Our algorithmic choices were motivated by simplicity: the method only relies on bit arrays and classical hash functions. While the idea of recording collisions in bit arrays to create MPHFs is not novel [6, 12], to the best of our knowledge *BBhash* is the first implementation that is competitive with the state of the art. The construction is particularly time-efficient as it is parallelized and mainly consists in hashing keys and performing memory accesses. Moreover, the additional data structures used during construction are provably small enough to ensure a low memory overhead during construction. In other words, creating the MPHF does not require much more space than the resulting MPHF itself. This aspect is important when constructing MPHFs on large key sets in practice.

Experimental results show that *BBhash* generates MPHFs that are slightly larger to those produced by other methods. However *BBhash* is by far the most efficient in terms of construction time, query time, memory and disk footprint for indexing large key sets (of cardinality above 10^9 keys). The scalability of our approach was confirmed by constructing MPHFs for sets as large as 10^{12} keys. To the best of our knowledge, no other MPHF implementation has been tested on that many keys.

A time/space trade-off is achieved through the γ parameter. The value $\gamma = 1$ yields MPHFs that occupy roughly $3N$ bits of space and have little memory overhead during construction. Higher γ values use more space for the construction and the final structure size, but they achieve faster construction and query times. Our results suggest that $\gamma = 2$ is a good time-versus-space compromise, using 3.7 bits per key. With respect to hypergraph-based methods [1, 3, 4, 11], *BBhash* offers significantly better construction performance, but the resulting MPHF size is up to 1 bit/key larger. We however argue that the MPHF size, as long

as it is limited to a few bits per key, is generally not a bottleneck as many applications use MPHFs to associate much larger values to keys. Thus, we believe that this work will unlock many high performance computing applications where the possibility to index billions keys and more is a huge step forward.

An interesting direction for future work is to obtain more space-efficient MPHFs using our method. We believe that a way to achieve this goal is to slightly change the hashing scheme. We would like to explore an idea inspired by the CHD algorithm for testing several hash functions at each level and selecting (then storing) one that minimizes the number of collisions. At the price of longer construction times, we anticipate that this approach could significantly decrease the final structure size.

Acknowledgments. We thank the GenOuest BioInformatics Platform that provided the computing resources necessary for benchmarking. We thank Djamal Belazzougui for helpful discussions and pointers.

References

1. Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. Cache-oblivious peeling of random hypergraphs. In *Data Compression Conference (DCC), 2014*, pages 352–361. IEEE, 2014.
2. Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.
3. Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Algorithms and Data Structures*, pages 139–150. Springer, 2007.
4. Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, 2013.
5. Chin-Chen Chang and Chih-Yang Lin. Perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005. doi:10.1093/comjnl/bxh074.
6. Jarrod A. Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P. Schroth, and Daniel S. Rokhsar. Meraculous: de novo genome assembly with short paired-end reads. *PloS one*, 6(8):e23501, 2011.
7. Yupeng Chen, Bertil Schmidt, and Douglas L Maskell. A hybrid short read mapping accelerator. *BMC Bioinformatics*, 14(1):67, 2013. doi:10.1186/1471-2105-14-67.
8. Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.
9. Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1):1–143, 1997.
10. Michael L. Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984.
11. Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In V. Andrew Goldberg and S. Alexander Kulikov, editors, *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, pages 339–352. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-38851-9_23.
12. Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*, pages 2774–2778. IEEE, 2006.
13. George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

14 Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 170–175. IEEE, 1982.

15 Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 746–755. Society for Industrial and Applied Mathematics, 2008.

16 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. *Retrieval and Perfect Hashing Using Fingerprinting*, pages 138–149. Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-07959-2_12.

A Proofs of MPHF size and memory required for construction

MPHF size with $\gamma = 1$.

$$\begin{aligned}\sum_{d\geq 0}|A_d| &= N\sum_{d\geq 0}(1-e^{-1})^d \\ &= N\frac{1}{1-(1-e^{-1})} \qquad \text{as } \lim_{d\to+\infty}(1-e^{-1})^d = 0 \\ &= eN\end{aligned}$$ ◀

MPHF size using any $\gamma \geq 1$. With $\gamma \geq 1 : |A_d| = \gamma|A_{d-1}|(1-e^{\frac{-1}{\gamma}}) = \gamma|A_0|(1-e^{\frac{-1}{\gamma}})^d = \gamma N(1-e^{\frac{-1}{\gamma}})^d$. Thus,

$$\sum_{d\geq 0}|A_d| = \gamma N\sum_{d\geq 0}(1-e^{\frac{-1}{\gamma}})^d\,.$$

Moreover, as $\lim_{d\to+\infty}(1-e^{\frac{-1}{\gamma}})^d = 0$ since for $\gamma > 0, 0 < 1-e^{\frac{-1}{\gamma}} < 1$, on has:

$$\sum_{d\geq 0}|A_d| = \gamma N\frac{1}{1-(1-e^{\frac{-1}{\gamma}})} = \gamma e^{\frac{1}{\gamma}}N\,.$$ ◀

Note that this proof stands for any γ value > 0, but that with $\gamma < 1$ the theoretical and practical MPHF sizes increase exponentially as γ get close to zero.

Proof of Lemma 1. Let $m(d)$ be memory required during level d and let R be the ratio between the maximal memory needed during the MPHF construction and the MPHF total size denoted by S. Formally,

$$R = \frac{\max_{d\geq 0}(m(d))}{S} = \frac{\max_{d\geq 0}(m(d))}{\gamma e^{\frac{1}{\gamma}}N}\,.$$

First we prove that $\lim_{d\to\infty}\frac{m(d)}{S} = 1$.

$$m(d) = \sum_{i<d}|A_i| + 2|A_d| = \gamma N\left(\frac{1-(1-e^{\frac{-1}{\gamma}})^d}{e^{\frac{-1}{\gamma}}} + 2(1-e^{\frac{-1}{\gamma}})^d\right)$$

Since for $\gamma > 0$, $0 < 1-e^{\frac{-1}{\gamma}} < 1$, then $\lim_{d\to\infty} m(d) = \gamma e^{\frac{1}{\gamma}}N$. Thus $\lim_{d\to\infty}\frac{m(d)}{S} = 1$.

Before going further, we need to compute $m(d+1) - m(d)$:

$$
\begin{aligned}
m(d+1) - m(d) &= \sum_{i<d+1} |A_i| + 2|A_{d+1}| - \sum_{i<d} |A_i| + 2|A_d| \\
&= |A_d| + 2|A_{d+1}| - 2|A_d| = 2|A_{d+1}| - |A_d| \\
&= 2\gamma N(1 - e^{\frac{-1}{\gamma}})^{d+1} - \gamma N(1 - e^{\frac{-1}{\gamma}})^d \\
&= \gamma N(1 - e^{\frac{-1}{\gamma}})^d (2(1 - e^{\frac{-1}{\gamma}}) - 1) \\
&= \gamma N(1 - e^{\frac{-1}{\gamma}})^d (1 - 2e^{\frac{-1}{\gamma}})
\end{aligned}
$$

We now prove $R \leq 1$ when $\gamma \leq \frac{1}{\log(2)}$ and also, $R < 2$ when $\gamma > \frac{1}{\log(2)}$.

- Case 1: $\gamma \leq \frac{1}{\log(2)}$.
  We have $\frac{m(0)}{S} = 2e^{-\frac{1}{\gamma}} \leq 2e^{-\log(2)} = 1$.
  Moreover, as $m(d+1) - m(d) = \gamma N(1 - e^{\frac{-1}{\gamma}})^d (1 - 2e^{\frac{-1}{\gamma}})$ and as, with $\gamma \leq \frac{1}{\log(2)}$: $1 - e^{\frac{-1}{\gamma}} \geq 0.5$, and $1 - 2e^{\frac{-1}{\gamma}} \geq 0$ then $m(d+1) - m(d) \geq 0$, thus, m is an increasing function.
  To sum up, with $\gamma \leq \frac{1}{\log(2)}$, we have **1**/ that $\frac{m(0)}{S} \leq 1$, **2**/ that $\lim_{d\to\infty} \frac{m(d)}{S} = 1$, and **3**/ that m is increasing, then $R \leq 1$.
- Case 2: $\gamma > \frac{1}{\log(2)}$.
  We have $\frac{m(0)}{S} = 2e^{-\frac{1}{\gamma}}$. With $\gamma > \frac{1}{\log(2)}$, $1 < \frac{m(0)}{S} < 2$. Moreover, $m(d+1) - m(d) = \gamma N(1 - e^{\frac{-1}{\gamma}})^d (1 - 2e^{\frac{-1}{\gamma}})$ is negative as: $1 - e^{\frac{-1}{\gamma}} > 0$ and $1 - 2e^{\frac{-1}{\gamma}} < 0$ for $\gamma > \frac{1}{\log(2)}$. Thus m is a decreasing function with d.
  With $\gamma > \frac{1}{\log(2)}$, we have **1**/ that $\frac{m(0)}{S} < 2$, /**2** that $\lim_{d\to\infty} \frac{m(d)}{S} = 1$ and /**3** that m is decreasing. Thus $R < 2$. ◀

B Algorithms pseudo-codes

Algorithm 1: `MPHF` construction.

Data: F_0 a set of N keys, integers γ and *last*
Result: array of bit arrays $\{A_0, A_1, \ldots, A_{last}\}$, hash table H

```
i=0;
while F_i not empty and i ≤ last do
    A_i = ArrayFill(F_i, γ);
    foreach key x of F_i do
        h = hash(x) mod (γ * N);
        if A_i[h] == 0 then
            F_{i+1}.add(x)
    i=i+1;
Construct H using remaining elements from F_{last+1};
Return {A_0, A_1, ..., A_last, H}
```

In practice F_i with $i > 1$ are stored on disk (see Section 3.4). The hash table H ensures that elements in F_{last+1} are mapped without collisions to integers in $[|F_0| - |F_{last+1}| + 1, |F_0|]$

Algorithm 2: *ArrayFill*

```
Data: F array of N keys, integer γ
Result: bit array A
Zero-initialize A and C two bit arrays with γ ∗ N elements;
foreach key x of F do
    h = hash(x) mod (γ ∗ N);
    if A[h] == 0 and C[h] == 0 then
        A[h] = 1;
    if A[h] == 1 and C[h] == 0 then
        A[h] = 0;
        C[h] = 1;
    if A[h] == 0 and C[h] == 1 then
        Skip;
Delete C;
Return A;
```

Note that the case $A[h] == 1$ and $C[h] == 1$ never happens.

Algorithm 3: `MPHF` query

```
Data: bit arrays {A_0, A_1, ..., A_last}, hash table H, key x
Result: integer index of x
i=0;
while i ≤ last do
    h = hash_i(x) mod A_i.size();
    if A_i[h] == 1 then
        return Σ_{j<i} |A_j| + rank(A_i[h]) ;
    i = i + 1;
return H[x] ;
```

Note, when x is not an element from the key set of the `MPHF`, the algorithm may return a wrong integer index.

C Commands

In this section we describe used commands for each presented result. Time and memory usages where computed using "*/usr/bin/time –verbatim*" unix command. The disk usage was computed thanks to a home made script measuring each 1/10 second the size of the directory using the "*du -sk*" unix command, and recording the highest value. The *BBhash* library and its *Bootest* tool are available from `https://github.com/rizkg/BBHash`.

C.1 Commands used for Section 4.1:

```
for ((gamma=1;gamma<11;gamma++)); do
./Bootest 1000000000 1 ${gamma} -bench
done
```

Note that 1000000000 is the number of keys tested and 1 is the number of used cores.

Additional tests, with larger key set and 8 threads:

```
for ((gamma=1;gamma<11;gamma++)); do
./Bootest 1000000000 1 ${gamma} -bench
done
```

C.2 Commands used for Section 4.2:

```
for keys in 10000000000 100000000000; do
./Bootest ${keys} 8 2 -bench
done
```

C.3 Commands used for Section 4.3:

We remind that our benchmark code, testing EMPHF, EMPHF MEM, CHD, and Sux4J is available at `https://github.com/rchikhi/benchmphf`.

- *BBhash* commands:

  ```
  for keys in 1000000 10000000 100000000 10000000000\
   10000000000 100000000000; do
  ./Bootest ${keys} 1 2 -bench
  done
  ```

- *BBhash* command with nodisk (Table 1) was

  ```
  ./Bootest 1000000000 1 2 -bench -nodisk
  ```

  and

  ```
  ./Bootest 1000000000 8 2 -bench -nodisk
  ```

  respectively for one and height threads. Other commands from Table 1 were deduced from previously presented *BBhash* computations.

- Commands EMPHF & EMPHF HEM:

  ```
  for keys in 1000000 10000000 100000000 10000000000\
   10000000000 100000000000; do
  ./benchmphf ${keys} -emphf
  done
  ```

  EMPHF (resp. EMPHF HEM) is tested by using the #define EMPHF_SCAN macro (resp. #define EMPHF_HEM). In order to assess the disk size footprint, the line "*unlink(tmpl);*" from file "*emphf/mmap_memory_model.hpp*" was commented.

- Commands CHD:

  ```
  for keys in 1000000 10000000 100000000 10000000000\
   10000000000 100000000000; do
  ./benchmphf ${keys} -chd
  done
  ```

- Commands Sux4J:
  for each size, the *"Sux4J/slow/it/unimi/dsi/sux4j/mph/LargeLongCollection.java"* was modified indicating the used size.

  ```
  ./run-sux4j-mphf.sh
  ```

C.4 Commands used for Section 4.4:

As explained Section 4.4, the keyString.txt file is composed of n-grams extracted from the Google Books Ngram dataset[2], version 20120701.

```
./BootestFile keyStrings.txt 10 2
```

[2] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

C.5 Commands used for Section 4.5:

BBhash command for indexing a trillion keys, with keys generated on the fly.

```
./Bootest 1000000000000 24 2 -onthefly
```

Generating Practical Random Hyperbolic Graphs in Near-Linear Time and with Sub-Linear Memory*

Manuel Penschuck

Goethe University, Frankfurt, Germany
mpenschuck@ae.cs.uni-frankfurt.de

Abstract

Random graph models, originally conceived to study the structure of networks and the emergence of their properties [8], have become an indispensable tool for experimental algorithmics. Amongst them, hyperbolic random graphs form a well-accepted family, yielding realistic complex networks while being both mathematically and algorithmically tractable. We introduce two generators MemGen and HyperGen for the $G_{\alpha,C}(n)$-model, which distributes n random points within a hyperbolic plane and produces $m = n\bar{d}/2$ undirected edges for all point pairs close by; the expected average degree $\bar{d}$ and exponent $2\alpha+1$ of the power-law degree distribution are controlled by $\alpha>1/2$ and C. Both algorithms emit a stream of edges which they do not have to store. MemGen keeps $\mathcal{O}(n)$ items in internal memory and has a time complexity of $\mathcal{O}(n \log\log n + m)$, which is optimal for networks with an average degree of $\bar{d} = \Omega(\log\log n)$. For realistic values of $\bar{d} = o(n/\log^{1/\alpha}(n))$, HyperGen reduces the memory footprint to $\mathcal{O}([n^{1-\alpha}\bar{d}^{\alpha} + \log n]\log n)$.

In an experimental evaluation, we compare HyperGen with four generators among which it is consistently the fastest. For small $\bar{d} = 10$ we measure a speed-up of 4.0 compared to the fastest publicly available generator increasing to 29.6 for $\bar{d} = 1000$. On commodity hardware, HyperGen produces $3.7{\cdot}10^8$ edges per second for graphs with $10^6 \leq m \leq 10^{12}$ and α=1, utilising less than 600 MB of RAM. We demonstrate nearly linear scalability on an Intel Xeon Phi.

1998 ACM Subject Classification G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases Random hyperbolic graph generator, streaming algorithm

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.26

1 Introduction

Even though most practical algorithms aim for a good performance on real-world data, artificial benchmarks are crucial for their development. Suited real-world datasets are typically scarce, do not scale, may exhibit noise or have uncontrollable properties. Tunable synthetic instances based on random models alleviate these issues. They are indispensable for systematic experiments allowing to quantify an algorithm's performance as a function of controllable parameters. Selecting the right model depends on the use case:

Many real-world networks (e.g., communication or social networks) exhibit basic features, such as a small diameter, a power-law degree distribution, and a non-vanishing local cluster coefficient [2, 3, 21, 23]. Amongst suited models, geometric random networks seem most natural. They explain the high local clustering of social networks[1] by embedding the nodes into a geometric space. Then the distance between any two nodes determines the probability of an edge between them. While Euclidean space is appropriate for spatial networks (e.g., [13]),

* Partially supported by the DFG grant ME 2088/3-2.
[1] I.e., a high triangle count expressing the intuition that two friends of a person are likely to acquaint too.

© Manuel Penschuck;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 26; pp. 26:1–26:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

it distorts complex networks, such as the internet graph, for which hyperbolic embedding (cf. Section 1.3) performs well [7, 24].

The task of actually generating instances of hyperbolic random graphs has been approached recently yielding generators that are either fast in practice [27] or optimal in theory [9]. We target the generation of large instances whose set of nodes[2] does not fit into memory. Space requirements are crucial especially in the context of co-processors with small dedicated memory. Another application of such a generator is the experimental evaluation of streaming [11, 20] or external memory algorithms [1, 22]. Since our algorithm is typically faster than the time it takes to write data to disk, one can connect it to the algorithm under testing without a round-trip to secondary storage. In such a case, the generator should leave the majority of memory to the main application in order to allow fast context switches.

1.1 Our contribution

We introduce two related generators MemGen (Section 2) and HyperGen (Section 3) for the $G_{\alpha,C}(n)$-model (Section 1.3) for large instances with n nodes and $m = \bar{d}n/2$ edges where $\bar{d}$ is the expected average degree. Both generators target a streaming setting and are compatible with the external memory model for practical instances. MemGen requires $\mathcal{O}(n)$ internal memory and has a time complexity of $\mathcal{O}(n \log\log n + m)$, which is optimal for networks with an average degree of $\bar{d}{=}\Omega(\log\log n)$. For realistic values of $\bar{d}{=}o(n/\log^{1/\alpha}(n))$, HyperGen reduces the memory footprint to $\mathcal{O}([n^{1-\alpha}\bar{d}^{\alpha}{+}\log n]\log n)$, where $\alpha{>}1/2$ controls the exponent $2\alpha{+}1$ of the result's power-law degree distribution. In an experimental evaluation (Section 5), HyperGen consistently the previously fastest generators we are aware of.

In the quest for a smaller memory footprint, we increase the data locality leading to an easily parallelisable algorithm. While we only explore shared memory parallelism, HyperGen works in a distributed setting with constant communication.

1.2 Notation

Define $[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_{>0}$. A graph $G = (V, E)$ has $n = |V|$ sequentially numbered nodes $V = \{v_i\}_{i\in[n]}$ and $m = |E|$ edges. Unless stated differently, graphs are undirected, unweighted, and have an average degree of $\bar{d} = 2m/n$. Let $N_G(v) \subseteq V$ be the neighbourhood of node v in graph G, i.e. the set of adjacent nodes.

We mainly consider points (r, θ) in polar coordinates where r is the radius (i.e. distance from the origin) and θ is the polar angle or azimuth. A point with radius r_1 is said to be *above* a point with radius r_2 if $r_1 > r_2$ and *below* if $r_1 < r_2$. Let $B_y(z)$ the ball of radius y and centre at radius z, i.e. the set of all points with distance at most y from point[3] z [12]. We apply standard set operations to balls where $\setminus$, $\cup$, and $\cap$ denote set differences, union and intersection accordingly.

Let $\mu(X)$ denote the probability mass of X. We denote a Binomial distribution over n items with probability p as $\mathcal{B}(n, p)$. *Also refer to Appendix A for a summary of definitions.*

1.3 The hyperbolic random graph model $G_{\alpha,C}(n)$

We consider the well-accepted $G_{\alpha,C}(n)$ model [12]. It follows the initial zero-temperature model of Krioukov et al. [16], but removes a redundant curvature parameter by setting $\zeta{=}1$.

[2] Implementations typically use at least 80 byte/node (cf. Section 5.2).
[3] We omit the azimuth of z as it is irrelevant in our analysis due to polar symmetry.

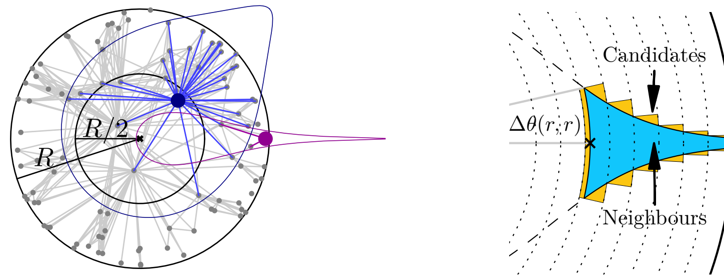


Figure 1 Left: The $G_{\alpha,C}(n)$ model with n=150, α=1, C=−2. The area enclosed by each coloured lobe corresponds to all points in distance at most R around its highlighted centre. **Right:** Band model introduced by NkGen (not to scale). The partial blue lobe indicates the area in which candidates can be found. The step-wise overestimation for candidate selection is shown in yellow.

▶ **Definition 1** (Gugelmann et al. [12]). Let $\alpha > 1/2$, $n \in \mathbb{N}_{>0}$, and $C > -2\log n$. The random graph $G_{\alpha,C}(n) = (V, E)$ has the following properties (cf. Figure 1):

- Each node $v_i \in V = \{v_1, \ldots, v_n\}$ is modelled by a random point $p_i = (r_i, \theta_i)$ in the hyperbolic plane.[4] Its angular coordinate θ_i is drawn uniformly from $[0, 2\pi]$ while its radius $0 \le r_i < R$ with $R := 2\log n{+}C$ is governed by the density function

$$\rho(r) = \frac{\alpha \sinh(\alpha r)}{\cosh(\alpha R) - 1}. \tag{1}$$

- The distance $d(p_i, p_j)$ between the two points p_i and p_j is given by

$$\cosh(d(p_i, p_j)) = \cosh(r_i)\cosh(r_j) - \sinh(r_i)\sinh(r_j)\cos(\theta_i - \theta_j). \tag{2}$$

Two nodes $v_i, v_j \in V$ are adjacent iff they have a small distance $d(p_i, p_j) < R$ and $i{\neq}j$.

Intuitively, the smaller α the more likely are points with small radial components, which are expected to have a high number of neighbours. The parameter hence controls the skewness of the resulting power-law degree distribution with an exponent of $\gamma = 2\alpha{+}1 > 2$ [16]. We assume $\alpha = \mathcal{O}(1)$ since real networks typically exhibit $2 \le \gamma \le 4$ (e.g., [10, 17]). Further, while with high probability there exists a giant component of linear size for $\alpha < 1$, networks with $\alpha > 1$ have components of sub-linear size [6]. The parameter C controls the average degree $\bar{d}$ of the graph which is governed as follows: [12]

$$\mathbb{E}\left[\,\bar{d}\,\right] = \frac{2}{\pi}\left(\frac{\alpha}{\alpha - 1/2}\right)^2 e^{-C/2}(1 + o(1)) \tag{3}$$

1.4 Hyperbolic graph generators

A naïve generator for hyperbolic graphs checks all $\binom{n}{2}$ pairwise distances and emits an edge for each pair of points close enough. On the one hand, such an approach can be implemented with constant memory overhead based on a pseudo-random hash function mapping node ids to coordinates. On the other, it incurs a sequential runtime of $\Theta(n^2)$ and is hence prohibitively expensive for large n. Similarly, while it can be fully parallelised yielding a

[4] We treat a node v_i and its corresponding point p_i as equivalent and use the terms interchangeably. Similarly, the symbols r_i and θ_i always refer to the radius and azimuth of point p_i.

$\mathcal{O}(1)$ time computation on a EREW-PRAM [14] with $p = \Theta(n^2)$ processors, such a solution requires $\Theta(n^2)$ work and is infeasibly inefficient.

All sub-quadratic algorithms we are aware of rather rely on a two-step approach: For each node $v \in V$, the generators firstly identify a set of candidates $C(v) \subseteq V$ by some geometric means (see below). Edges are then generated by computing only the distances between v and $C(v)$. In order to avoid false negatives, all neighbours $N(v)$ have to be a subset of $C(v)$. Techniques include:

- Looz et al. [26] project all points into the Poincaré disk model which allows neighbourhood queries based on Euclidean disks. Candidates are selected using a polar quad-tree. The authors bound the generator's runtime to $\mathcal{O}((n^{3/2} + m)\log n)$ with high probability.
- Later, Looz et al. improve the runtime significantly by dropping the angular separation of the quad-tree [27]. As sketched in Figure 1, their generator (which is the basis of our work and to which we refer as NKGEN[5]) decomposes the hyperbolic plane into $k{=}\Theta(\log n)$ bands, each covering the radial range $[b_j, b_{j+1})$ where $b_j = (1 - \beta^{j-1})R/(1 - \beta^k)$ for $j \in [k{+}1]$ and a tuning parameter $\beta{\approx}0.9$. In a preprocessing step, the points are randomly scattered over the plane by inserting each point (r, θ) into the appropriate band j, where $b_j \le r < b_{j+1}$. The points are then sorted by their angular coordinates independently for each band.
  In order to query the neighbour candidates of a point $p{=}(r, \theta)$ stored in band i, the algorithm iterates over all bands $i \le j \le k$. For each band j, it computes the angular range $A_j = [\theta - \Delta\theta(r, b_j), \theta + \Delta\theta(r, b_j)]$ where the maximal angular distance $\Delta\theta(r, b_j)$ between p and any hypothetical point in band j is given by
  $$\Delta\theta(r,b) := \begin{cases} \pi & \text{if } r + b < R \\ \operatorname{acos}\left[(\cosh(r)\cosh(b) - \cosh(R))/(\sinh(r)\sinh(b))\right] & \text{otherwise} \end{cases}. \quad (4)$$
  The points within A_j constitute all candidates from band j. Since points are sorted by their angular coordinates, the bounds of A_j can be identified using two binary searches in time $\mathcal{O}(\log n)$. The authors experimentally find a runtime of $\mathcal{O}(n \log n + m)$.
- Bringmann et al. [9] propose the *Geometric Inhomogeneous Random Graph* model (GIRG) and show that $G_{\alpha,C}(n)$ is a special case of GIRG which can be generated with their sampling algorithm in expected time $\mathcal{O}(n + m)$. Their sampling method for hyperbolic graphs is similar to the quad-tree approach in the sense that it partitions the space uniformly along the angular axis and exponentially in the radial direction. The resulting cells roughly correspond to leaves in a quad-tree. However, the algorithm does not execute fine-grained neighbourhood queries for each node; it rather tests all point pairs of two related cells in a pessimistic and data-oblivious fashion. Despite its expected linear runtime, the algorithm seems to suffer from high constants (cf. Section 5). Bläsius et al. provide an implementation[6] to which we refer as GIRGGEN [5].
- Very recently and independently from this work S. Lamm proposed a communication-agnostic distributed generator RHGEN with a partitioning scheme similar to [9], although with different radial limits [18]. Each band is split into disjoint buckets of equal angular size. Their number is chosen such that each cell is expected to contain k points, where $k \approx 4$ is a tuning parameter. RHGEN allows all processing units to compute the points within any bucket independently, eliminating the need of communication. The author

[5] A reference implementation is included in NetworKit [25], `https://networkit.iti.kit.edu/`.
[6] `https://bitbucket.org/HaiZhung/hyperbolic-embedder/overview`

Algorithm 1: MemGen

```
   Input  : Number of nodes n, Radius of bounding circle R, Density α, Spacing β
1  Δθ(a,b) := π if a+b < R else acos[(cosh(a) cosh(b) − cosh(R))/(sinh(a) sinh(b))];
2  noBands ← max(2, ⌈βR⌉);
3  limits ← [0, R/2, c+R/2, 2c+R/2, ..., R−c, R] with c = R/2/(noBands − 1);
4  for i ∈ [1, ..., n] do
5  |  r ← random radius from [0, R) with density ρ(r) = α sinh(αr)/(cosh(αR) − 1);
6  |  b ← search band s.t. limits[b] ≤ r < limits[b + 1];
7  |  θ ← next non-decreasing uniformly random polar angle;
   |  // In case θ + 2Δθ(r,r) > 2π special treatment is necessary -
   |     cf. text
8  |  bands [b].addPoint(Point(i, (r, (θ+Δθ(r,r)) mod 2π)));
9  |  b ← max(2, b);
10 |  bands[b].addRequest(Request(i, [r, r+2Δθ(r, max(r, limits[b+1]))], (r, θ+Δθ(r,r))))
   // Main Phase:  Generation of Edges
11 foreach u, v ∈ bands[1].points with u < v do
12 |  emit edge {u.id, v.id};
13 reqsToAbove ← [];
14 for b ∈ [2, ..., noBands] do
15 |  sort bands [b].points by angle;
16 |  reqsFromBelow ← sorted(reqsToAbove);
17 |  initialise empty reqsToAbove, candidates;
18 |  foreach pt ∈ bands[b].points do
19 |  |  remove all requests from candidates ending before pt.θ;
20 |  |  foreach req ∈ (bands[b].reqs ∪ reqsFromBelow) with req.rangeBegin ≥ pt.θ do
21 |  |  |  insert req into candidates if not existing;
22 |  |  |  insert req into reqsToAbove with updated range;
23 |  |  foreach req ∈ candidates do
24 |  |  |  if (req.r, req.id) ≤_lexico (pt.r, pt.id) ∧ dist(pt, req) ≤ R then
25 |  |  |  |  emit edge {pt.id, req.id};
```

shows an expected sequential runtime of $\mathcal{O}(n+m)$, bounds the generation time of the distributed grid structure to $\mathcal{O}(P \log n + n/P)$, where P is the number of processors, and empirically finds a time-complexity of $\mathcal{O}(\frac{n+m}{P} + P \log n)$ for the parallel algorithm.

2 MemGen: a fast algorithm with linear memory usage

To simplify the description of HyperGen and present its main design, we start with a sequential version MemGen (cf. Alg. 1) requiring $\mathcal{O}(n)$ memory. Most arguments regarding the runtime of this algorithm will later translate into the space complexity bound of HyperGen.

Geometrically, MemGen employs a band partitioning similar to the one introduced by NkGen and illustrated in Figure 1. However, we alter their contents and access patterns, and use different radial band limits: all bands except the lowest one have a constant height $x = R/2k$, where $k{+}1$ is the number of bands and $x = \Theta(1)$ a tuning parameter

(typically $x \in [1, 2]$). Band $1 \leq i \leq k{+}1$ covers a radial range of $[l_i, l_{i+1})$ with $l_1 = 0$ and $l_i = [1 + (i{-}2)/k] \cdot R/2$ for some $k{=}\Theta(R)$. It is not necessary to further divide the lowest band since all points with radius $r \leq R/2$ are forming a clique (cf. Figure 1) and can be handled without vicinity tests.

Band b stores all points contained. For each point p within b or below it, the band additionally maintains a so-called *request* $\texttt{req}_b(p)$, storing the coordinates of p itself as well as the angular range in which neighbours of p can lie in band b. Such requests effectively reduce random accesses during the candidate selection and carry pre-computed values repeatedly required for the distance calculations (cf. Section 4).

In fact, the algorithm chooses a request-centric view and randomly draws the beginnings of each request range, computes its radius-dependent length, and then places a point at its centre.[7] We draw the polar components as sorted random numbers using the online technique detailed in [4] requiring constant time per element. The generation process may yield requests with a range $[a, b]$ with $b > 2\pi$. To take the azimuthal 2π-period of the hyperbolic disk into account, we split such queries into two separate ranges $[0, b{-}2\pi]$ and $[a, 2\pi]$ respectively and mark the latter as a copy. Analogously, points with $\theta > 2\pi$ are remapped to $\theta - 2\pi$. After the generation phase, the points are sorted by their polar coordinate.

In the main phase, we iterate over the bands starting from the centre for which we simply emit the clique of all nodes contained. For all higher bands, we scan through the points and requests in lock-step and keep a separate list of candidates $C(\cdot)$. Since both streams are sorted, we can efficiently update $C(v)$ when moving from one point to the next.

Each time we reach a new unmarked request $\texttt{req}_j(p)$, we propagate it to the next higher band $j{+}1$ by adding $\texttt{req}_{j+1}(p)$ to the appropriate insertion buffer. Here, it may be again necessary to split a request due to the 2π-periodicity. Further observe that the range of a request may shrink during the propagation. As a consequence, the insertion buffer has to be sorted when switching to band $j{+}1$ (cf. Section 2.2) before it can be merged with the requests generated in the preprocessing phase. In a last step, we compute the distance between a point and all candidates in order to emit the edges.

The linear time generators we are aware of use discrete buckets along the angular axis to avoid sorting [9, 18]. However, preliminary experiments with MemGen suggested that a more involved candidate selection process is faster in practice (especially in the context of vectorisation) and does incur only small theoretical penalties (cf. Theorem 7). Thus, we maintain a data structure which keeps active candidates in a continuous array to facilitate vectorisation efficiently (cf. Section 4). The array has an arbitrary order allowing to implement deletions as moves of the array's back. The data structure is further augmented with a search tree to find the position of a candidate using its point id as key. We also keep a priority queue with range-ends to quickly find and remove obsolete candidates.

2.1 Candidate selection is at worst a constant approximation

In this section, we establish all necessary facts to show that the candidate selection incurs a non-substantial overhead. In Lemma 2, we will see that most points issue only a constant number of requests.

Subsequently, we derive a high-probability bound on the number of candidates processed for any node in two steps: Observe that a node has to process all requests from nodes below. Lemma 3 bounds their number in terms of n and average degree $\bar{d}$. Further, Lemma 4 states

---

[7] While this is an arbitrary choice for MemGen, it will become a crucial ingredient for HyperGen.

that MemGen overestimates the probability mass during candidate selection by at most a constant factor. Therefore, the bound on the number of neighbours from below carries over to the number of candidates processed.

▶ **Lemma 2.** *The expected number of bands* $\mathbb{E}[B_i]$ *a random node* v_i *sends requests to is* $\mathbb{E}[B_i] = 1 + \frac{1-e^{-\alpha R/2}}{e^{\alpha/2k}-1} = \mathcal{O}(1)$ *where* $k{+}1 = \Theta(R)$ *is the number of bands used by* MemGen.

Proof. Each point with radius r sends requests to its own band j with $b_j{\le}r{<}b_{j+1}$ as well as to all above. Consequently, the probability of a random point p_i contributing to band j is governed by the mass function $\mu(B_{b_{j+1}}(0))$ as given by Eq. (21). Using indicator variables for the reception of a request by band j, we obtain the claimed expectation value:

$$\mathbb{E}[B_i] = \sum_{j=0}^{k} \mu(B_{b_{j+1}}(0)) = \sum_{j=0}^{k} e^{\alpha[\frac{R}{2}(1+j/k)-R]} = e^{-\alpha R/2} \sum_{j=0}^{k} \left(e^{\alpha\frac{R}{2k}}\right)^j = 1 + \frac{1-e^{-\alpha R/2}}{e^{\alpha/2k}-1}\,. \blacktriangleleft$$

▶ **Lemma 3.** *Let* N_j *be the number of neighbours the point* $p_j{=}(r_j,\theta_j)$ *has from below, i.e. neighbours with smaller radius. With high probability, there exist* $\mathcal{O}(n/\log^2 n)$ *points with* $N_j = \mathcal{O}(n^{1-\alpha}\bar{d}^{\alpha}\log(n))$ *while the remainder of points with* $r_j > R/2$ *has* $N_j = \mathcal{O}(n^{1-2\alpha}\log^{2\alpha}(n)\bar{d}^{2\alpha})$ *neighbours.*

Proof. Let $X_1,\ldots,X_n$ be indicator variables with $X_i{=}1$ if p and p_i are adjacent. Due to radial symmetry we directly obtain the expectation value of X_i conditioned on the radius p_i:

$$\mathbb{E}[X_i \mid r_i{=}\,x] = \mathbb{P}[X_i{=}1 \mid r_i{=}\,x] = \begin{cases} 1 & \text{if } x < R-r \\ \Delta\theta(x,r)/\pi & \text{otherwise} \end{cases} \tag{5}$$

We remove the conditional using the Law of Total Expectation and equations (20) and (21):

$$\mathbb{E}[X_i] = \int_0^{R-r} \rho(x)\mathrm{d}x \quad + \quad \frac{1}{\pi}\int_{R-r}^{r} \rho(x)\Delta\theta(x,R)\mathrm{d}x \tag{6}$$

$$= \left[e^{-\alpha r}{-}e^{-\alpha R}\right](1{+}o(1)) + \frac{1}{\pi}\frac{\alpha}{\alpha{-}\frac{1}{2}}e^{-\alpha r}\left[e^{(\alpha-\frac{1}{2})(2r-R)}-1\right](1\pm\mathcal{O}(e^{-r})) \tag{7}$$

Fix the radius $r_T = R{-}\frac{2}{\alpha}\log\log n$ with $R/2 < r_T$ (wlog) and consider three cases for r:

- We ignore all points $r \le R/2$ as they belong to the central clique and are irrelevant here.
- Observe that with high probability there exist $\mathcal{O}(n/\log^2(n))$ points below r_T. Exploiting the monotonicity of Eq. 7 in r, we maximise it by setting $r = R/2$, which cancels out the second term. Linearity of the expectation value, substitution of $R = 2\log(n) + C$, and Eq. (3) yield $\mathbb{E}[\sum_i X_i] = \mathcal{O}\left[n\left(\bar{d}/n\right)^{\alpha}\right]$. Then, Chernoff's bound gives $\sum_i X_i = \mathcal{O}(n^{1-\alpha}\bar{d}^{\alpha}\log(n))$ with high probability.
- For all points above r_T, set $r = r_T$ yielding $\sum_i X_i = \mathcal{O}(n^{1-2\alpha}\log^{2\alpha}(n)\bar{d}^{2\alpha})$ with high probability analogously. ◀

▶ **Lemma 4.** *Consider a query point with radius* r *and a band with boundaries* $[a,b)$. MemGen*'s candidate selection overestimates the probability mass of the actual query range by a factor of* $OE(b{-}a,\alpha)$ *where* $OE(x,\alpha) := \frac{\alpha-1/2}{\alpha}\frac{1-e^{\alpha x}}{1-e^{(\alpha-1/2)x}}$.

Proof. If $r < R{-}b$, the requesting point covers the band completely which renders the candidate selection process optimal. We now consider $r \ge R{-}a$ and omit the fringe case of $R{-}b < r < R{-}a$ which follows analogously (and by continuity between the two other cases).

Then, the probability mass μ_Q of the intersection of the actual query circle $B_R(r)$ with the band $B_b(0) \setminus B_a(0)$ is given by

$$\mu_Q := \mu\left[(B_b(0) \setminus B_a(0)) \cap B_R(r)\right] \tag{8}$$

$$= \mu\big[\big(B_R(0) \cap B_R(r)\big) \setminus B_a(0)\big] - \mu\big[\big(B_R(0) \cap B_R(r)\big) \setminus B_b(0)\big] \tag{9}$$

$$\overset{(22)}{=} \frac{2\alpha e^{-\frac{r}{2}}}{\pi(\alpha - \frac{1}{2})} \left[\left(1 + \frac{\alpha - \frac{1}{2}}{\alpha + \frac{1}{2}} e^{-2\alpha b}\right) e^{(\alpha - \frac{1}{2})(b-R)} + \left(1 + \frac{\alpha - \frac{1}{2}}{\alpha + \frac{1}{2}} e^{-2\alpha a}\right) e^{(\alpha - \frac{1}{2})(a-R)} \right] (1+\epsilon) \tag{10}$$

$$= \frac{2}{\pi} e^{-\frac{r}{2} - (\alpha - \frac{1}{2})R} \left[\frac{\alpha}{\alpha - \frac{1}{2}} \left(e^{(\alpha - \frac{1}{2})b} - e^{(\alpha - \frac{1}{2})a} \right) + \mathcal{O}\left(e^{-(\alpha - \frac{1}{2})a} \right) \right], \tag{11}$$

where ϵ substitutes the error term expanded in the last line (cf. Figure 1, blue cover of a band).

MemGen overestimates the actual query range at the border and covers the mass μ_H (see Figure 1, yellow cover of a band):

$$\mu_H := \frac{1}{\pi} \Delta\theta(r, a) \int_a^b \rho(y) dy = \frac{1}{\pi} \cdot 2e^{\frac{R-a-r}{2}} (1 + \mathcal{O}(e^{R-a-r})) \cdot \frac{\cosh(\alpha b) - \cosh(\alpha a)}{\cosh(\alpha R) - 1} \tag{12}$$

$$= \frac{2}{\pi} e^{\frac{R-a-r}{2}} (1 + \mathcal{O}(e^{R-a-r})) \cdot \left[e^{\alpha(b-R)} - e^{\alpha(a-R)} \right] (1 \pm o(1)) \tag{13}$$

$$= \frac{2}{\pi} e^{-\frac{r}{2} - (\alpha - \frac{1}{2})R} \left[e^{\alpha b - a/2} - e^{(\alpha - \frac{1}{2})a} \right] \cdot \left(1 \pm \mathcal{O}\left(e^{(1-\alpha)(R-a)-r} \right) \right) \tag{14}$$

The claim follows by the division of both mass functions μ_H/μ_Q. ◀

▶ **Corollary 5.** *Given a constant band height, i.e. $b - a = \mathcal{O}(1)$, Lemma 4 implies a constant overestimation for any $\alpha > 1/2$. In case of $b - a = 1$, we have $OE(1, \alpha) \le \sqrt{e} \approx 1.64 \;\; \forall \alpha > 1/2$.*

2.2 Nearly sorted points/request allow for faster sorting

MemGen's scheme to update the candidate list requires the input streams of requests and points to be increasing in their angular coordinate. Since we are not aware of a technique that directly yields both in an ordered fashion, we have to sort them. Using naïve methods this would amount to $\mathcal{O}(n \log(n))$ time (cf. Lemma 2). Since the number $m = n\bar{d}/2$ of edges generated constitutes a lower bound on the time complexity of any generator, this approach is optimal for $\bar{d} = \Omega(\log n)$.

Observe, however, that the points are calculated based on ordered requests and are therefore already nearly sorted. Similarly, requests have to be sorted after being propagated from ordered streams. In both cases, and with high probability, the change of rank of each item is bounded to some $\Delta = o(n)$.[8] Such a Δ-ordered sequence can be sorted in time $\mathcal{O}(n \log \Delta)$, e.g. using a sliding window coupled with a priority queue of size Δ.

The following Lemma gives a rough bound on the time complexity which suffices to show that MemGen is optimal for $\bar{d} = \Omega(\log\log(n))$ with high probability:

▶ **Lemma 6.** *Sorting all points initially and requests after their propagation requires $\mathcal{O}\left(n \min[\log(\bar{d} \log n), \log n]\right)$ time.*

[8] Split requests and remapped points are sorted separately and merged in linear time.

Proof. It suffices to bound the claim for requests since every point contributes at least one request and has a shorter lifetime. As stated in the introduction of the Lemma, we can rely on classical sorting in time $\mathcal{O}(n \log n)$ for the case of $\bar{d} = \Omega(\log n)$. Thus assume $\bar{d} = o(\log n)$.

The proof consists then of two steps: We pick a radius r_T, s.t. with high probability there are only $\mathcal{O}(n/\log^2(n))$ points below r_T. Since each point issues at most $\mathcal{O}(\log n)$ requests, we can classically sort their $\mathcal{O}(n/\log(n))$ tokens in time $\mathcal{O}(n)$. For the remaining points, we bound the number of overlapping requests from above and thereby also the maximal change in rank that can occur during sorting.

The number n_T of points below radius r_T is governed by the Binomial distribution $\mathcal{B}(n, B_{r_T}(0))$ with $B_{r_T}(0) = 1/\log^2(n)$. Solving for r_T yields $r_T = R - \frac{2}{\alpha} \log\log n$ and hence $n_T = \mathcal{O}[n B_{r_T}(0)]$ with high probability.

We now tend to the requests above r_T and exploit the two following facts:

- The number of bands above r_T is constant since $r_T/R \to 1$ as $n \to \infty$.
- During sorting only those requests that overlap can change their relative position. Therefore, we fix $\theta \in [0, 2\pi)$ and let n_θ be the number of requests that include θ.

To maximise n_θ, assume without loss of generality that all remaining requests lie at radius r_T. Then, n_θ is binomially distributed around its mean $n\mu$ with[9]

$$n\mu = n\frac{\Delta\theta(r_T, r_T)}{\pi} = 2ne^{-\frac{R}{2} + \frac{2}{\alpha}\log\log n} = \mathcal{O}\left(\bar{d}\log^{\frac{2}{\alpha}}(n)\right). \tag{15}$$

With high probability only $\mathcal{O}\left(\bar{d}\log^{\frac{2}{\alpha}}(n)\right)$ requests overlap due to Chernoff's inequality. We thus can sort them in time $\mathcal{O}(n \log(\bar{d} \log n))$. ◀

▶ **Theorem 7.** *MemGen requires $\mathcal{O}(n)$ memory and has a runtime of $\mathcal{O}(n \log\log n + m)$ with high probability.*

Proof. The space complexity directly follows from Alg. 1: each of the n points is stored in exactly one band, yields at most two requests, and requires $\mathcal{O}(1)$ space in the candidate list. During the main phase, there further exists only one insertions buffer at a time to which a point may contribute $\mathcal{O}(1)$ items. △

We bound MemGen's time complexity by considering each component individually:

- The preprocessing (until line 10) requires $\mathcal{O}(1)$ time per point making it non-substantial.
- Handling of cliques is trivially bounded by $\mathcal{O}(m)$ since every iteration emits an edge.
- The sorting steps (lines 15 and 16) require $\mathcal{O}(n \log(\bar{d} \log n)) = \mathcal{O}(n \log \bar{d} + n \log\log n)$ time in total with high probability according to Lemma 6.[10]
- By applying Lemma 3 and Cor. 5, the candidate selection requires $\mathcal{O}(n \log \bar{d})$ time with high probability.
- All distance calculations require in total $\mathcal{O}(m)$ time since Cor. 5 bounds the fraction of computations that do not yield an edge to $\mathcal{O}(1)$. ◀

3 HyperGen: reducing MemGen's memory footprint

In the analysis of MemGen, we repeatedly exploited the facts that requests are generated in increasing angular order and the majority affects only a small fraction of the hyperbolic

[9] It can be improved to $\mathcal{O}(\bar{d} \log\log n)$ by replacing the assumption that all requests lay at r_T with an appropriate integral; we omit this non-substantial calculation in favour of simplicity.

[10] We consider only the first min-term: In case the second term becomes smaller, the theorem's claim is dominated by the $\mathcal{O}(m)$ where $m = n\bar{d}/2$.

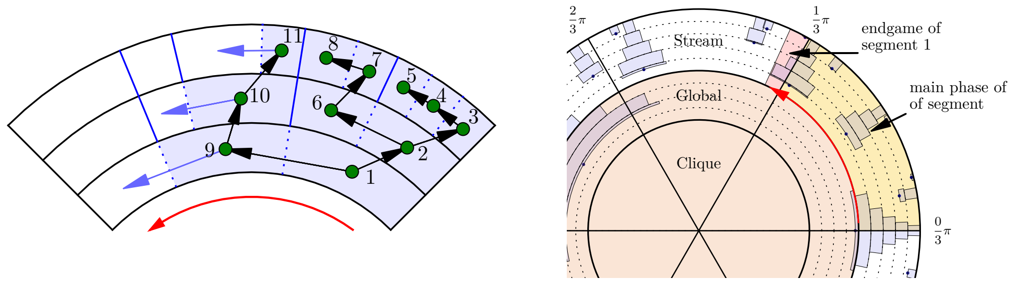


Figure 2 Left: HyperGen streams through each band consuming batches whose size is limited by two factors: either due to a polar limit imposed by the underlying band (solid blue line) or due to the limited number of requests a batch is allowed to have (dotted blue line). We traverse the indicated tree in depth-first order. **Right**: The hyperbolic plane is partitioned along the polar axis into p segments of equal size. Radially, there are two groups: the lower *global bands* which are preprocessed and kept in memory, and the upper *streaming bands*. In the main phase, each execution thread streams through its segment towards increasing polar angles (red arrow). Requests overlapping into the next segment are then completed in the endgame.

plane. This is also the foundation of HyperGen, which strives to additionally reduce the memory requirements of the generator. In order to do so, we do not draw all points globally and insert them into their bands, but rather reverse the scheme.

HyperGen first computes how many points go into each band. It is then able to draw points for each band independently. Due to the radial distribution function $\rho(r)$, band i with boundaries $[l_i, l_{i+1})$ carries a probability mass of $\mu_i = \mu[B_{l_{i+1}}(0) \setminus B_{l_i}(0)]$. Consequently, the numbers $N = (n_1, \ldots, n_k)$ of points per band with $n = \sum_i n_i$ are governed by a multinomial distribution with μ_i as event probabilities. We sample N and build for each band i a stream $S_i(n_i, s_i)$ that outputs exactly n_i requests with monotonously increasing angles as detailed in Section 2. Storing the seed value s_i used to initialise the underlying pseudo-random number generator enables HyperGen to replay the stream from the beginning.

Analogous to MemGen, each band maintains such a request stream S_i, the current candidates, and a small list of recently produced points. The generator starts with the innermost band $i = 1$ (cf. optimisation in Section 3.1) and draws a batch of at most c requests from its stream S_i, computes the positions of their corresponding points, and finally sorts the latter by their angle. Let θ_L be the beginning of the last request generated (θ_L=2π if the batch is empty). We merge the newly generated points with those remaining from the band's last batch, update the set of candidates, and match points against them as described for MemGen. Edges produced are pushed into the output stream.

Before we continue in the current band i, we first process all higher bands, hence limiting the amount of requests in memory. HyperGen propagates the recently generated requests to the band i+1. Observe that the request of a point (r, θ) is always centred around θ but its range shrinks as it is moved to higher bands. As a direct consequence, the higher range is completely enclosed by the lower one and no future request produced for band i will ever start before θ_L. Therefore, we recurse to band i+1 but limit processing there to points with $\theta < \theta_L$. In effect, HyperGen performs a depth-first traversal of the recursion tree illustrated in Figure 2 in which every node corresponds to a batch.

Due to the processing limit imposed on higher bands, we make sure they have the same information they would receive in MemGen. One subtle difference, however, concerns the fact that MemGen splits requests and remaps points overlapping the 2π threshold to take their angular periodicity into account. This is not possible in HyperGen since overlaps in outer bands are only detectable quite late in a run.

We resolve this issue by ignoring it at first, i.e. we perform the main computation phase exactly as described above. If there are still pending candidates or points after its completion, we restart the request streams to handle the so-called *endgame*. During endgame, HyperGen executes the same algorithm as before but only emits edges for pairs in which either the point or the request originate from the main phase. Therefore, it can be stopped as soon as all such *old* points and candidates have been processed. A single rewind suffices and thus does not affect the asymptotic runtime since a request has a length of at most 2π rad and a point can only be moved π rad in forward direction.

▶ **Theorem 8.** *For $c=\mathcal{O}(1)$ HyperGen requires $\mathcal{O}([n^{1-\alpha}\bar{d}^{\alpha}+\log n]\log n)$ memory with high probability, where $\bar{d}$ is the expected average degree and n the number of nodes.*

Proof. Each of the $k=\Theta(\log n)$ bands requires auxiliary data structures of constant size. Regarding the data contained, it again suffices to show the result for requests (cf. proof of Lemma 6). The number of points N_C with radius below $R/2$ is governed by a binomial distribution $\mathcal{B}(n,\mu(B_{R/2}(0)))$. Thus, with high probability $N_C=\mathcal{O}(n^{1-\alpha}\bar{d}^{\alpha}+\log n)$ where the second term ensures concentration for small $(\bar{d}/n)^{\alpha}$. Each such point contributes requests to $k=\mathcal{O}(\log n)$ bands; multiplication yields the claim.

According to Lemma 3 and Cor. 5 and for any fixed θ, there are with high probability $\mathcal{O}(n^{1-\alpha}\bar{d}^{\alpha}\log n)$ points with radius $r\geq R/2$ that have at least one request including θ. By Lemma 2, they contribute to $\mathcal{O}(1)$ bands on average and thus are covered by the claim. ◀

▶ **Corollary 9.** *In the external memory model with $M=\Omega([1+n^{1-\alpha}\bar{d}^{\alpha}]\log n)$, HyperGen only triggers I/Os to write out the resulting m edges in $\mathcal{O}(scan(m))$ I/Os.*

3.1 Accelerating the Endgame

A runtime/memory trade-off can be implemented to improve the runtime (especially in the context of the parallel variant). Rather than starting the streaming approach introduced above, we compute all bands with radii at most r_G and store them as in MemGen in the so-called *global phase*. This allows us to propagate split requests to the streaming bands which in turn allows us to stop the endgame earlier.

Observe that a request of a point (r_G,θ) has a length of at most $2\Delta\theta(r_G,r_G)$. To restrict the endgame to a fraction $1/f$ of the hyperbolic plane, we solve $2\Delta\theta(r_G,r_G)=2\pi/f$ for r_G. The number $n_G(f)$ of points generated in the global phase, which have to be kept in internal memory, is thus binomially distributed around the mean of

$$\mathbb{E}\left[n_G(f)\right]=n\mu(B_{r_G}(0))=n\left(\frac{\bar{d}f}{2n}\right)^{\alpha}\left(\frac{\alpha-\frac{1}{2}}{\alpha}\right)^{2\alpha}=\mathcal{O}\left(n^{1-\alpha}\bar{d}^{\alpha}f^{\alpha}\right). \tag{16}$$

3.2 Parallelism

Similarly to NkGen, HyperGen can easily be parallelised by decomposing the hyperbolic plane into p segments of equal size along the polar axis. As shown in Figure 2, we use a global phase with $f\geq p$ to handle the n_G requests spanning more than one segment. We enqueue a copy of each such request into all segments it affects. For realistic settings, it suffices to execute this phase sequentially; however, parallelism can be applied as in NkGen's implementation. The number of points in each segment $(\nu_1,\ldots,\nu_p)$ with $n-n_G=\sum_i\nu_i$ is then sampled from a multinomial distribution in which each event is equally likely. Based on this distribution, each band continues independently as described in the original formulation

of HyperGen. In the endgame, each segment retrieves the seed values of its successor's pseudo-random number generators and replays its streams.

In a distributed scenario the seed values can be computed using a pseudo-random hash function mapping the segment id to a pseudo-random seed value. Further, the initial distribution as well as the fast global phase can be computed repeatedly by each compute node, yielding constant communication.

4 Implementation

The prototypical implementation is available at `https://github.com/manpen/hypergen/`.

4.1 Adjacency tests without trigonometric functions

In a preliminary study we found that NkGen's runtime is dominated by trigonometric computations during the calculation of distances between points and their neighbour candidates. We approach this issue by introducing a new pre-computing scheme inspired by the usage of the Poincaré disk model in [26]. We project the random points into the unit disk causing additional work per point but simplifying all further distance computation. Thus, the speed-up increases with the average degree.

Our implementation applies the transform only to the distance calculations and does not change the candidate selection process. Let $p{=}(r_p, \theta_p)$ and $q{=}(r_q, \theta_q)$ be two points in the hyperbolic space and $p' = (\mathrm{cdm}(r_p), \theta_p)$ and $q' = (\mathrm{cdm}(r_q), \theta_q)$ their counterparts in the Poincaré disk model, where $\mathrm{cdm}(r) := [(1 - r^2)/(1 + r^2)]^{1/2}$. Then p and q are adjacent if

$$R > d(p', q') = \operatorname{acosh}\left(1 + 2\frac{||q-p||^2}{(1-||p||^2)(1-||q||^2)}\right) \tag{17}$$

$$\Leftrightarrow \quad \frac{\cosh(R)-1}{2} > \frac{||q-p||^2}{(1-||p||^2)(1-||q||^2)} = \frac{(x_{p'}-x_{q'})^2 + (y_{p'}-y_{q'})^2}{(1-r_{q'}^2)(1-r_{q'}^2)} \tag{18}$$

$$= \left((x_{p'}-x_{q'})^2 + (y_{p'}-y_{q'})^2\right) \cdot \gamma(r_{p'}) \cdot \gamma(r_{q'}), \tag{19}$$

where $x_{p'}{=}r_{p'}\sin(\theta_p)$ and $y_{p'}{=}r_{p'}\cos(\theta_p)$ are the Cartesian coordinates of point p' (analogously for q'). We reduce a distance computation to three additions and four multiplications by pre-computing $x_{p'}, y_{p'}$ and $\gamma(r_{p'}) := 1/(1 - r_{p'}^2)$ for each point. The resulting expression can be vectorised effectively and even allows to partially fuse operations (cf. FMA instructions).

Our implementation uses explicit vectorisation[11] only during the distance computation. For graphs with small average degree, a speed-up may be possible by vectorising per-point computations such as the random number generation and geometric transformations.

4.2 Optimising NkGen for streaming

In addition to the default implementation of NkGen, we study a variant NkGenOpt to which we apply the following optimisations:[12]

[11] Based on libVC – SIMD Vector Classes for C++ [15], `https://github.com/VcDevel/Vc`.
[12] NkGen originally generates an adjacency-list-like internal-memory data-structure using the NetworKIT's GraphBuilder module. This limits the graph sizes and explains NkGen optimisation for smaller graphs. Further, the removal of the GraphBuilder in this work shifts the implementation's balance and leads to the large optimisation potential demonstrated. Porting the optimisations back to NetworKIT showed insignificant changes for typical instances which could be likely solved with an optimised GraphBuilder.

- It avoids recalculations similar to Section 4.1, but does not rely on the Poincaré transform. In NkGen's case all additional data has to be kept in memory amounting to roughly 32 bytes per points. We expect that this increase is only significant for very sparse graphs as NetworKit keeps the whole adjacency list in RAM.
- The number of binary searches as well as their range[13] is reduced. Further, the amount of data copied is significantly decreased also resulting in less (de-)allocation operations. This optimisation roughly compensates the increased footprint due to the pre-computations.
- We removed several checks which are not required for the restricted case of $G_{C,\alpha}(n)$.

HyperGen and NkGenOpt are verified against NkGen over a wide range of parameters. Here, we observed only acceptable numerical discrepancies for large graphs affecting less than one edge out of 10^5, caused by the different implementations of the distance computation.

5 Experimental evaluation

In this section, we compare six configurations: HyperGen on CPU / Xeon Phi (cf. Section 3), NkGen [27], NkGenOpt (cf. Section 4.2), RHGen [18], and GirgGen [5]. They are implemented in C++ and built as release versions using the same compiler. As an exception, HyperGen has to use a hardware-specific compiler, links against Intel's TBB malloc_proxy, but otherwise has the same code basis as the CPU version.

To fully exploit HyperGen's on-the-fly edge generation, none of the implementations writes the edge list into memory. We rather simulate a very simplistic streaming algorithm which consumes the edge stream and computes a fingerprint by summing all node indices contained.[14] This choice enforces that the generators have to compute and forward every edge but does not impose memory restrictions. With the exception of GirgGen, all generators support parallelism and are configured to use all available hardware threads. RHGen employs a multi-process design using MPI allowing several compute nodes, while HyperGen, NkGen and NkGenOpt use lightweight threads based on OpenMP.

The runtime benchmarks were conducted on one of the following systems:

- Indicated by (Phi): Intel Xeon Phi 5120D (60 cores, 240 threads, 1.05GHz), 8 GB GDDR5 RAM Linux 2.6.38, ICC 17.0.0, Intel TBB malloc_proxy
- Otherwise: Intel Xeon CPU E5-2630 v3 (8 cores, 16 threads, 2.40GHz) with AVX2/SSE4.2 support for 4-way double-precision vectorisation, 64 GB 2133 MHz RAM, Linux 4.8.1, GCC 6.2.1, VC (8. Dec. 2016), MPICH 3.2-7

The number of repetitions per data point (with different random seeds) is denoted by S. All plots show the median of repeated measurements and errorbars corresponding to the unbiased estimation of the standard deviation. Due to its large runtime GirgGen typically only includes one measurement per data point.

5.1 Runtime

We study the generators' runtimes for a wide range of graph sizes. For each run, we fix the number of nodes $10^5 \leq n \leq 10^9$ as well as the average target degree $\bar{d} \in \{10, 1000\}$, which we consider as lower and upper limits of realistic inputs [3, 19, 21, 23]. In order

[13] By replacing $\Delta\theta(r, b_i)$ by $\Delta\theta(r, r)$ when searching candidates for point (r, θ) in band i with $b_i \leq r < b_{i+1}$.
[14] We removed the appropriate memory allocations and accesses from NkGen, RHGen, and GirgGen, and added the streaming simulation. The patches are included in our repository.

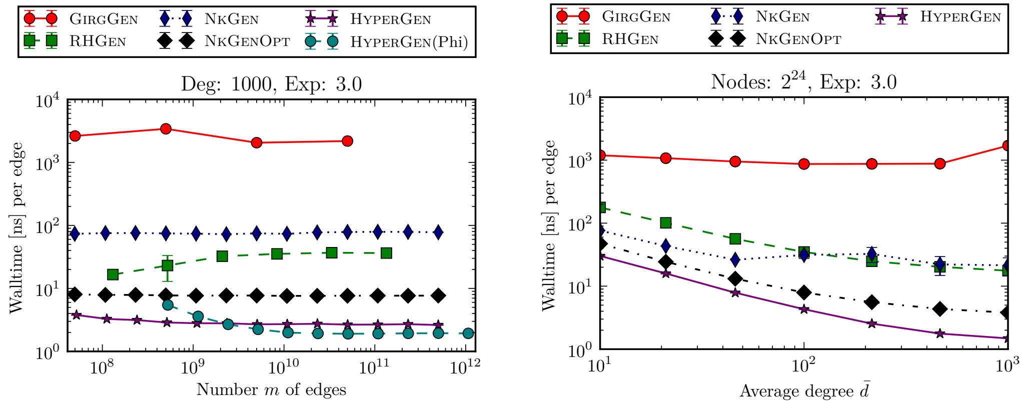


Figure 3 Runtime/edge generated for α=1 (power-law exp. γ=3) as a function of n and $\bar{d}$. S=5.

to achieve compatible results, all implementations use values of R derived with NkGen's `getTargetRadius`-method. In case of HyperGen, we use two segments per thread to balance load for large average degrees. For RHGen we chose an expected bucket size of four which resulted in the best performance in preliminary tests.

As shown in Figures 3 and 6 (Appendix) and Table 1 (Appendix), HyperGen is consistently the fastest generator, followed by NkGenOpt which outperforms NkGen. GirgGen is always the slowest. If we assume perfect parallelisability and divide GirgGen's walltime by the number of cores, it is on par with NkGen for small degrees but remains up to one order of magnitude slower for $\bar{d} = 1000$. For $\bar{d} = 10$ NkGen outperforms RHGen, while for $\bar{d} = 1000$ and $\alpha = 1$ the opposite is true.

All generators but HyperGen (Phi) exhibit an almost constant computation time per edge for large n. The improvements of HyperGen (Phi) towards larger n can be attributed to the very high number of threads ($p = 240$) which incur more overhead compared to runs performed on a CPU. This overhead is amortised only for high values of n.

Based on Figure 6 (Appendix), we measure a speed-up of 4.0 for $\bar{d}$=10 and 29.6 for $\bar{d}$=1000 when comparing HyperGen to NkGen for $n \geq 10^8$ and $\alpha = 1$. Similar results for smaller n are included in Table 1 (Appendix). On (Phi), HyperGen is 2.3 times faster ($\bar{d}$=10) compared to the execution on the more modern CPU-based reference system. The speed-up reduces to 1.2 for $\bar{d}$=1000 which seems to be caused by a smaller cache per thread.

When using HyperGen to test a multi-pass streaming algorithm, it is virtually always faster to repeatedly regenerate the graph than to buffer it in external memory.

5.2 Memory consumption

The memory consumption is measured for the same parameter settings as above. We consider the maximal resident set size (i.e. the peak allocation of the generator) as reported by the operating system. While all implementations seem to have potential for further savings, Figures 4 and 7 (Appendix) show a clear trend: With the exception of HyperGen, all generators seem to converge to a linear growth for large n requiring ≈ 80 byte per node. RHGen exhibits higher constants which may be partially caused by overheads due to its MPI architecture spawning independent processes rather than lightweight threads and preventing cheap shared-memory utilisation.

Consistent with our analysis, HyperGen exhibits a sub-linear footprint rendering it orders of magnitude cheaper for large n. As the number n of nodes increases (and hence R

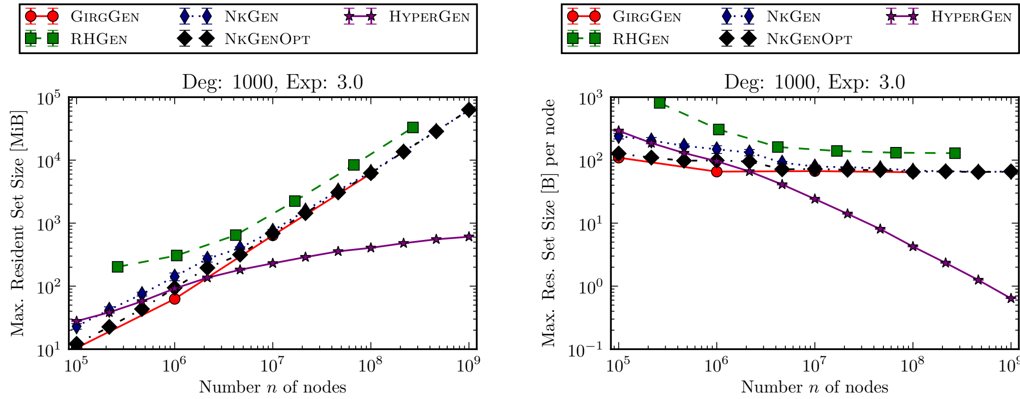


Figure 4 Maximal memory allocated during execution as measured by `time` for $\alpha = 1$.

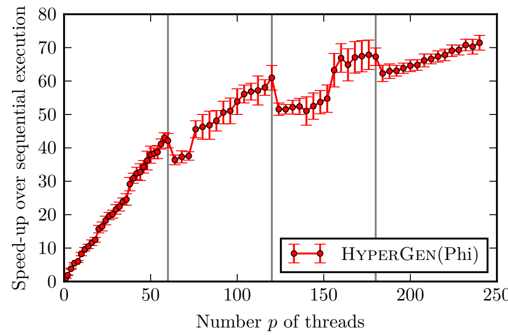


Figure 5 Strong scaling of HyperGen on (Phi) for a graph with n=10^8 and $\bar{d} = 10$. $S = 8$. Each vertical division marks a new level of HyperThreading.

for fixed $\bar{d}$), more points lie in the outer bands. Thus, a smaller fraction of points has to be handled (and stored) during the global phase. For the same reason, the memory footprint decreases with increasing α. To support Theorem 8 and the analysis in Section 3.1, we carried out additional runs up to n=10^{11} whose memory footprint is well within the noise observed for n=10^8. We do not include measurements for (Phi) since the memory allocation scheme adopted for the high number of threads does not yield meaningful set sizes.

5.3 Scalability

We measure HyperGen's scalability using strong scaling experiments on (Phi). This processor features 60 physical cores each offering four virtual threads (HyperThreading). While fixing the graph instance to n=10^8 and $\bar{d} = 10$, we record the runtime for an increasing number p of threads. As illustrated in Figure 5, the implementation exhibits a nearly linear speed-up of 43.0 ± 1.5 when utilising $p = 58$ threads. Surpassing this point, the computational power provided by the hardware does not scale linearly any more. Thus, the additional speed-up is less pronounced peaking at 71.4 ± 6 for $p = 240$.

Acknowledgments. The author thanks Ulrich Meyer, Kamil René König, Moritz von Looz and Alexander Schickedanz for valuable discussions and suggestions, Sebastian Lamm for providing the code and support for RHGen, Ivan Kisel and Egor Ovcharenko for their help

with the Xeon Phi, as well as the anonymous reviewers for their insightful comments and recommendations.

References

1. Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM, 31(9)*, pages 1116–1127, 1988.
2. Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *CoRR*, cond-mat/0106096, 2001. doi:10.1103/RevModPhys.74.47.
3. Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In *Web Science 2012, WebSci'12, Evanston, IL, USA – June 22-24, 2012*, pages 33–42, 2012. doi:10.1145/2380718.2380723.
4. Jon Louis Bentley and James B. Saxe. Generating sorted lists of random numbers. *ACM Trans. Math. Softw.*, 6(3):359–364, 1980. doi:10.1145/355900.355907.
5. Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. In *24th Annual European Symposium on Algorithms, ESA 2016, Aarhus, Denmark*, 2016. doi:10.4230/LIPIcs.ESA.2016.16.
6. Michel Bode, Nikolaos Fountoulakis, and Tobias Müller. *On the giant component of random hyperbolic graphs*, pages 425–429. Scuola Normale Superiore, Pisa, 2013. doi:10.1007/978-88-7642-475-5_68.
7. Marián Boguñá, Fragkiskos Papadopoulos, and Dmitri Krioukov. Sustaining the internet with hyperbolic mapping. *Nature Communications*, Sep 2010. doi:10.1038/ncomms1063.
8. Béla Bollobás. *Random Graphs*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2 edition, 2001. doi:10.1017/CBO9780511814068.
9. Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *CoRR*, abs/1511.00576, 2015. URL: http://arxiv.org/abs/1511.00576.
10. Sergei N Dorogovtsev and José FF Mendes. *Evolution of networks: From biological nets to the Internet and WWW*. OUP Oxford, 2013.
11. Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi, editors. *Data Stream Management – Processing High-Speed Data Streams*. Data-Centric Systems and Applications. Springer, 2016. doi:10.1007/978-3-540-28608-0.
12. Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: Degree sequence and clustering – (extended abstract). In *Automata, Languages, and Programming – 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, pages 573–585, 2012. doi:10.1007/978-3-642-31585-5_51.
13. Piyush Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Trans. Information Theory*, 46(2):388–404, 2000. doi:10.1109/18.825799.
14. Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
15. Matthias Kretz. *Extending C++ for explicit data-parallel programming via SIMD vector types*. PhD thesis, Goethe University Frankfurt am Main, 2015.
16. Dmitri V. Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Phys. Rev. E*, 82:036106, Sep 2010. doi:10.1103/PhysRevE.82.036106.
17. Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 611–617, 2006. doi:10.1145/1150402.1150476.
18. Sebastian Lamm. Communication efficient algorithms for generating massive networks. Master's thesis, Karlsruhe Institute of Technology, 2017. doi:10.5445/IR/1000068617.
19. Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

20 Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014. doi:10.1145/2627692.2627694.

21 Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web – revisited: a trick of the heavy tail. In *23rd International World Wide Web Conference, WWW'14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 427–432, 2014. doi:10.1145/2567948.2576928.

22 Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.

23 Seth A. Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy J. Lin. Information network or social network?: the structure of the twitter follow graph. In *23rd International World Wide Web Conference, Seoul, Republic of Korea*, 2014. doi:10.1145/2567948.2576939.

24 Yuval Shavitt and Tomer Tankel. Hyperbolic embedding of internet graph for distance estimation and overlay construction. *IEEE/ACM Trans. Netw.*, 16(1):25–36, 2008. doi:10.1145/1373452.1373455.

25 Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. doi:10.1017/nws.2016.20.

26 Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. Generating random hyperbolic graphs in subquadratic time. In *Algorithms and Computation – 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, pages 467–478, 2015. doi:10.1007/978-3-662-48971-0_40.

27 Moritz von Looz, Mustafa Safa Özdayi, Sören Laue, and Henning Meyerhenke. Generating massive complex networks with hyperbolic geometry faster in practice. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–6, 2016. doi:10.1109/HPEC.2016.7761644.

A Definitions, useful identities and approximations

A.1 Hyperbolic functions

$$\sinh(x) := \frac{1}{2}\left(e^{x} - e^{-x}\right) \qquad \operatorname{asinh}(y) = x \;\Rightarrow\; \sinh(x) = y$$

$$\cosh(x) := \frac{1}{2}\left(e^{x} + e^{-x}\right) \qquad \operatorname{acosh}(y) = x \;\Rightarrow\; \cosh(x) = y$$

A.2 Geometry related definitions

$$\rho(r) := \alpha\frac{\alpha\sinh(\alpha r)}{\cosh(\alpha R)} \qquad \text{radial density, cf. Eq 1}$$

$$\mu(B_r(0)) := \int_0^r \rho(x)\mathrm{d}x = \frac{\cosh(\alpha x) - 1}{\cosh(\alpha R)} \qquad \text{radial cdf}$$

$$\Delta\theta(r,b) := \begin{cases} \pi & \text{if } r{+}b < R \\ \operatorname{acos}\left[\frac{\cosh(r)\cosh(b) - \cosh(R)}{\sinh(r)\sinh(b)}\right] & \text{otherwise} \end{cases} \qquad \text{cf. Eq. 4}$$

A.3 Approximations

Gugelmann et al. derived the following approximations[15] [12]:

$$\Delta\theta(r,b) = \begin{cases} \pi & \text{if } r + b < R \\ 2e^{\frac{R-r-y}{2}}(1 + \Theta(e^{R-r-y})) & \text{if } r + b \geq R \end{cases} \tag{20}$$

$$\mu(B_r(0)) = \int_0^r \rho(x)\mathrm{d}x = \frac{\cosh(\alpha r)}{\cosh(\alpha R) - 1} = e^{\alpha(r-R)}(1 + o(1)) \tag{21}$$

$$\mu\left[(B_R(r) \cap B_R(0)) \setminus B_x(0)\right] = \frac{2}{\pi}\frac{\alpha e^{-r/2}}{\alpha - \frac{1}{2}} \cdot$$

$$\begin{cases} 1 \pm \mathcal{O}(e^{-(\alpha-\frac{1}{2})r} + e^{-r}) & \text{if } x < R{-}r \\ \left[1{-}(1{+}\frac{\alpha-\frac{1}{2}}{\alpha+\frac{1}{2}}e^{-2\alpha x})e^{-(\alpha-\frac{1}{2})(R-x)}\right](1 \pm \mathcal{O}(e^{-r}{+}e^{-r-(\alpha-\frac{3}{2})(R-x)})) & \text{if } x \geq R{-}r \end{cases} \tag{22}$$

---

[15] We drop the $(1 + \mathcal{O}(\cdot))$ error terms in our calculations without further notice if they are either irrelevant or dominated by other simplifications made

B Additional experimental results

Table 1 Comparison of generators for $n = 2^{26}$, $\alpha \in \{0.55, 1\}$, and $\bar{d} \in \{10, 1000\}$. *Comp* refers to the number of distance computations between two points. It does not include node pairs that could be ruled out earlier (e.g., by comparing indices or radii). For HyperGen the value is higher due to vectorisation which often prevents such early discarding. *RSS* is the maximal resident set size (i.e. peak memory allocation) as reported by the operating system. In case of RHGen it is the sum of RSS of all MPI processes yielding a higher overhead. GirgGen is a purely sequential implementation and includes fewer data points due to the high runtime. We report the standard deviation of the S measurements as uncertainty and apply statistical error propagation.

† Experiment was cancelled after a runtime of 10^5 s.

$n{=}2^{26}$, $\bar{d}{=}10$, $\alpha{=}0.55$, $R{=}39.2$			in total			per edge		relative to HyperGen		
Algo	S	Degree	Comp. $[10^8]$	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HyperGen	6	10.3 ± 0.4	7.5 ± 0.2	0.0 ± 0.0	11.8 ± 0.1	2.1 ± 0.1	34.0 ± 1.4	1	1	1
NkGen	6	9.8 ± 0.7	5.6 ± 0.3	4.6 ± 0.3	57.1 ± 3.0	1.7 ± 0.2	173 ± 22	0.8 ± 0.1	290 ± 22	4.8 ± 0.3
NkGenOpt	6	9.6 ± 0.4	5.3 ± 0.2	4.1 ± 0.0	36.0 ± 0.3	1.7 ± 0.1	111.7 ± 6.1	0.7 ± 0.0	263.5 ± 3.2	3.1 ± 0.0
RHGen	4	8.3 ± 0.1	7.9 ± 0.0	6.7 ± 0.6	110.0 ± 1.0	2.8 ± 0.0	395.8 ± 6.7	1.1 ± 0.0	428 ± 39	9.3 ± 0.1
GirgGen	3	10.0 ± 0.0	18.1 ± 0.0	3.9 ± 0.0	884.3 ± 0.8	5.4 ± 0.0	2635.5 ± 2.8	2.4 ± 0.1	248.1 ± 2.2	75.0 ± 0.5

$n{=}2^{26}$, $\bar{d}{=}10$, $\alpha{=}1.00$, $R{=}33.3$			in total			per edge		relative to HyperGen		
Algo	S	Degree	Comp. $[10^8]$	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HyperGen	5	9.7 ± 0.0	7.0 ± 0.0	0.0 ± 0.0	12.9 ± 0.1	2.1 ± 0.0	39.6 ± 0.3	1	1	1
NkGen	5	10.0 ± 0.0	5.5 ± 0.0	4.1 ± 0.0	54.9 ± 0.5	1.6 ± 0.0	163.5 ± 1.6	0.8 ± 0.0	602 ± 13	4.3 ± 0.1
NkGenOpt	5	10.0 ± 0.0	5.2 ± 0.0	4.1 ± 0.0	34.4 ± 0.2	1.6 ± 0.0	102.3 ± 0.8	0.8 ± 0.0	596 ± 12	2.7 ± 0.0
RHGen	5	10.0 ± 0.0	8.1 ± 0.0	7.5 ± 0.5	120.5 ± 0.4	2.4 ± 0.0	359.2 ± 1.2	1.2 ± 0.0	1100 ± 99	9.4 ± 0.1
GirgGen	3	10.0 ± 0.0	16.6 ± 0.0	3.9 ± 0.0	819.8 ± 7.1	5.0 ± 0.0	2443 ± 21	2.4 ± 0.0	566 ± 11	63.6 ± 1.0

$n{=}2^{26}$, $\bar{d}{=}1000$, $\alpha{=}0.55$, $R{=}29.5$			in total			per edge		relative to HyperGen		
Algo	S	Degree	Comp. $[10^8]$	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HyperGen	5	1052.2 ± 1.6	622.8 ± 1.2	0.2 ± 0.0	86.9 ± 0.4	1.8 ± 0.0	2.5 ± 0.0	1	1	1
NkGen	5	994.4 ± 3.3	456.2 ± 1.3	6.4 ± 0.3	955.5 ± 4.9	1.4 ± 0.0	28.6 ± 0.2	0.7 ± 0.0	27.2 ± 1.3	11.0 ± 0.1
NkGenOpt	5	991 ± 19	441.6 ± 7.8	4.2 ± 0.0	299.5 ± 5.1	1.3 ± 0.0	9.0 ± 0.3	0.7 ± 0.0	17.6 ± 0.3	3.4 ± 0.1
RHGen	5	889.1 ± 2.2	426.0 ± 1.1	23.9 ± 2.4	2205 ± 64	1.4 ± 0.0	73.9 ± 2.3	0.7 ± 0.0	101 ± 11	25.4 ± 0.9
GirgGen	1	1000.0	1160.6	3.8	55756.0	3.5	1661.6	1.9 ± 0.0	16.2 ± 0.1	641.5 ± 2.8

$n{=}2^{26}$, $\bar{d}{=}1000$, $\alpha{=}1.00$, $R{=}24.1$			in total			per edge		relative to HyperGen		
Algo	S	Degree	Comp. $[10^8]$	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HyperGen	5	1015.8 ± 1.3	616.2 ± 0.7	0.1 ± 0.0	84.3 ± 0.5	1.8 ± 0.0	2.5 ± 0.0	1	1	1
NkGen	5	999.9 ± 0.6	443.3 ± 0.3	4.4 ± 0.1	1878 ± 468	1.3 ± 0.0	56 ± 14	0.7 ± 0.0	43.7 ± 2.4	22.3 ± 5.7
NkGenOpt	5	999.7 ± 0.4	428.5 ± 0.2	4.2 ± 0.0	261.1 ± 6.7	1.3 ± 0.0	7.8 ± 0.2	0.7 ± 0.0	41.6 ± 1.1	3.1 ± 0.1
RHGen	5	999.1 ± 0.0	410.5 ± 0.2	8.1 ± 0.2	1234.8 ± 5.8	1.2 ± 0.0	36.8 ± 0.2	0.7 ± 0.0	79.8 ± 3.7	14.6 ± 0.2
GirgGen†	1				$\geq 10^5$					≥ 1150

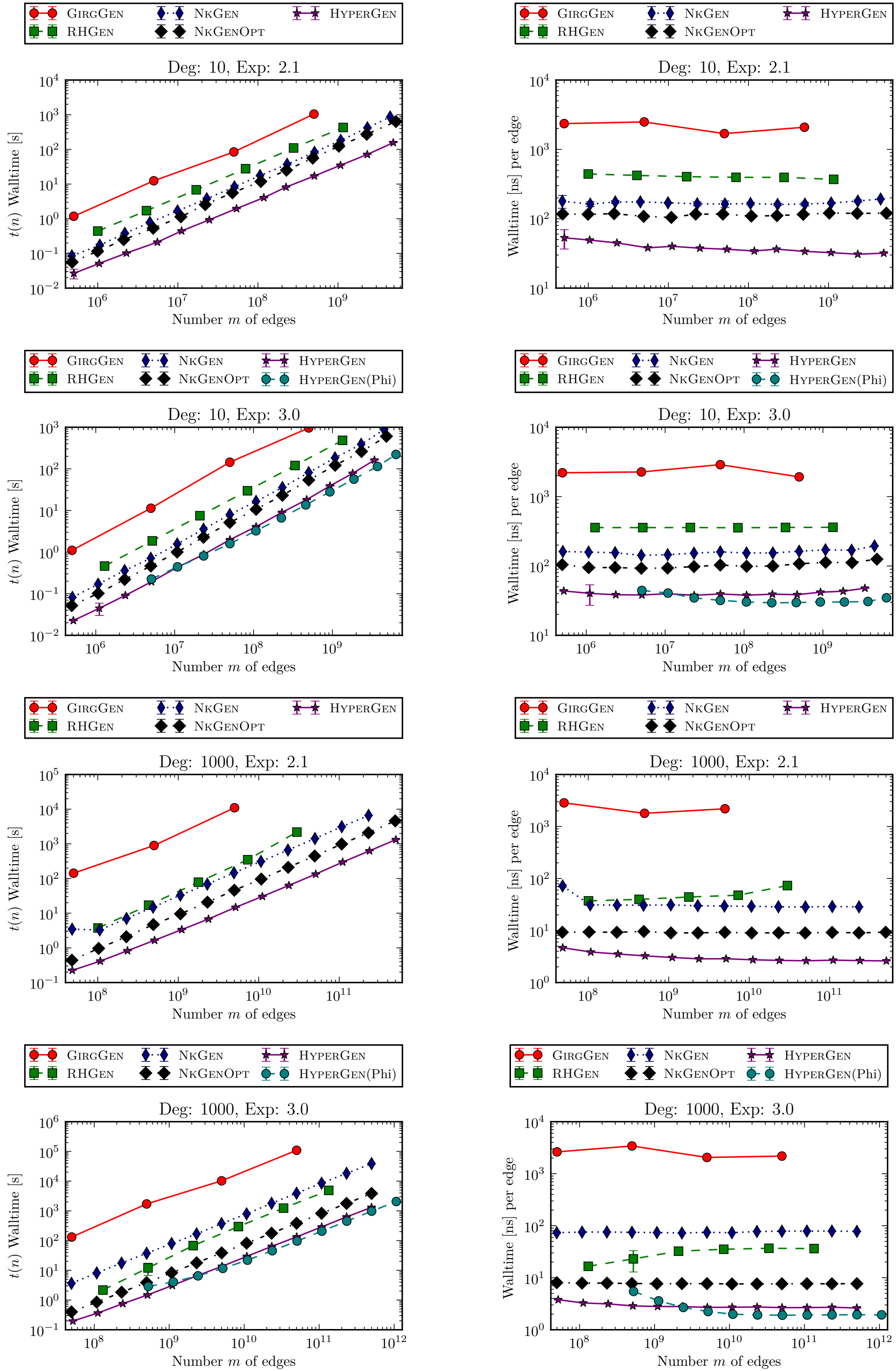


Figure 6 Runtime of generators as function of the number n of nodes.

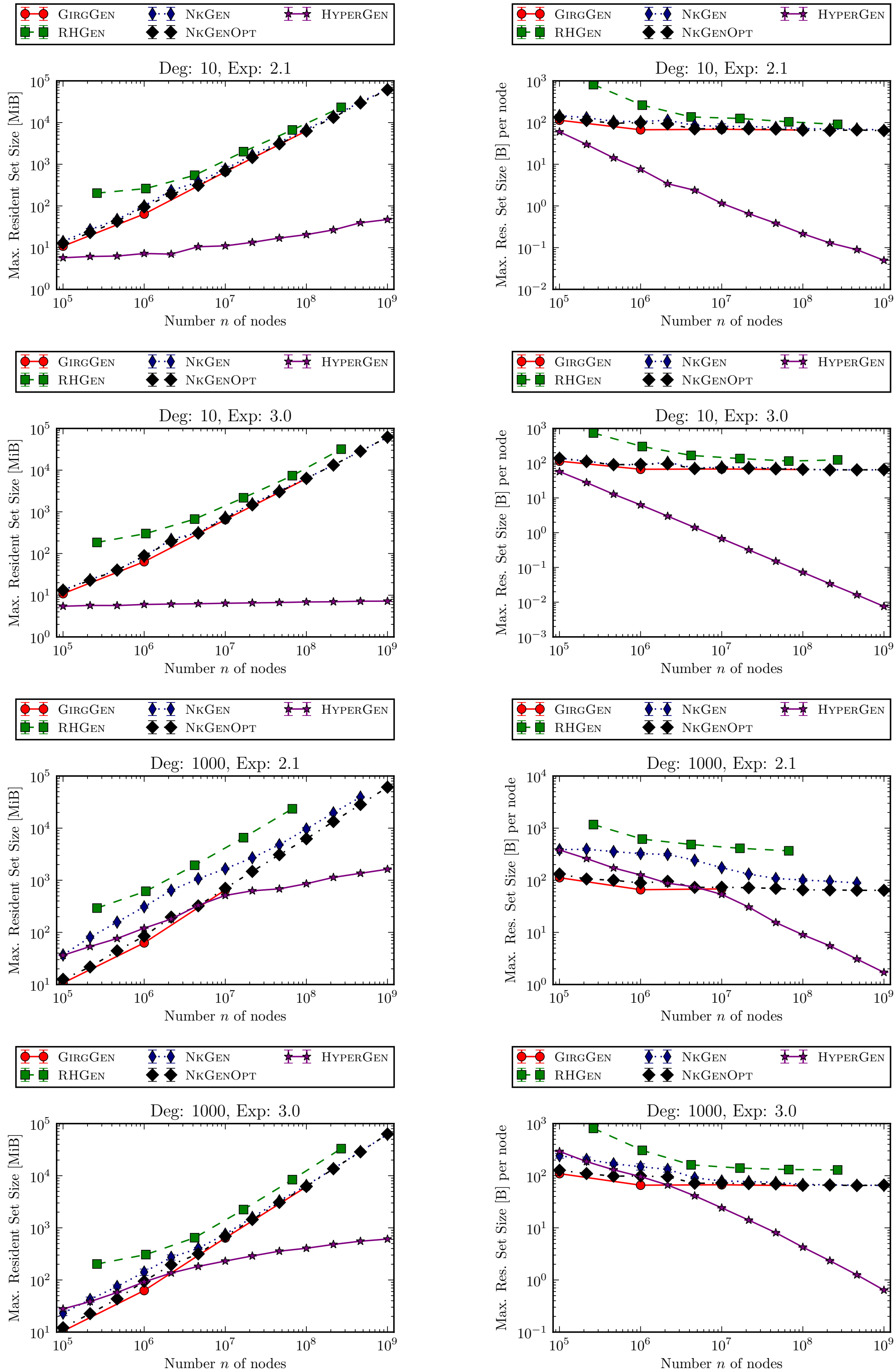


Figure 7 Max. memory allocation of generators as function of the number n of nodes.

Incremental Low-High Orders of Directed Graphs and Applications*

Loukas Georgiadis[1], Konstantinos Giannis[2], Aikaterini Karanasiou[3], and Luigi Laura[4]

1 **Department of Computer Science & Engineering, University of Ioannina, Ioannina, Greece**
`loukas@cs.uoi.gr`
2 **Department of Computer Science & Engineering, University of Ioannina, Ioannina, Greece**
`giannis_konstantinos@outlook.com`
3 **Università di Roma “Tor Vergata”, Rome, Italy**
`aikaranasiou@gmail.com`
4 **“Sapienza” Università di Roma, Rome, Italy**
`laura@dis.uniroma1.it`

Abstract

A flow graph $G = (V, E, s)$ is a directed graph with a distinguished start vertex s. The dominator tree D of G is a tree rooted at s, such that a vertex v is an ancestor of a vertex w if and only if all paths from s to w include v. The dominator tree is a central tool in program optimization and code generation, and has many applications in other diverse areas including constraint programming, circuit testing, biology, and in algorithms for graph connectivity problems. A low-high order of G is a preorder δ of D that certifies the correctness of D, and has further applications in connectivity and path-determination problems.

In this paper we consider how to maintain efficiently a low-high order of a flow graph incrementally under edge insertions. We present algorithms that run in $O(mn)$ total time for a sequence of edge insertions in a flow graph with n vertices, where m is the total number of edges after all insertions. These immediately provide the first incremental *certifying algorithms* for maintaining the dominator tree in $O(mn)$ total time, and also imply incremental algorithms for other problems. Hence, we provide a substantial improvement over the $O(m^2)$ straightforward algorithms, which recompute the solution from scratch after each edge insertion. Furthermore, we provide efficient implementations of our algorithms and conduct an extensive experimental study on real-world graphs taken from a variety of application areas. The experimental results show that our algorithms perform very well in practice.

1998 ACM Subject Classification E.1 [Data Structures] Graphs and Networks, Lists, Stacks, and Queues, Trees, G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases connectivity, directed graphs, dominators, dynamic algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.27

1 Introduction

A *flow graph* $G = (V, E, s)$ is a directed graph (digraph) with a distinguished start vertex $s \in V$. A vertex v is *reachable* in G if there is a path from s to v; v is *unreachable* if no

* A full version of the paper is available at `http://arxiv.org/abs/1608.06462`.

© Loukas Georgiadis, Konstantinos Giannis, Aikaterini Karanasiou, and Luigi Laura;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 27; pp. 27:1–27:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

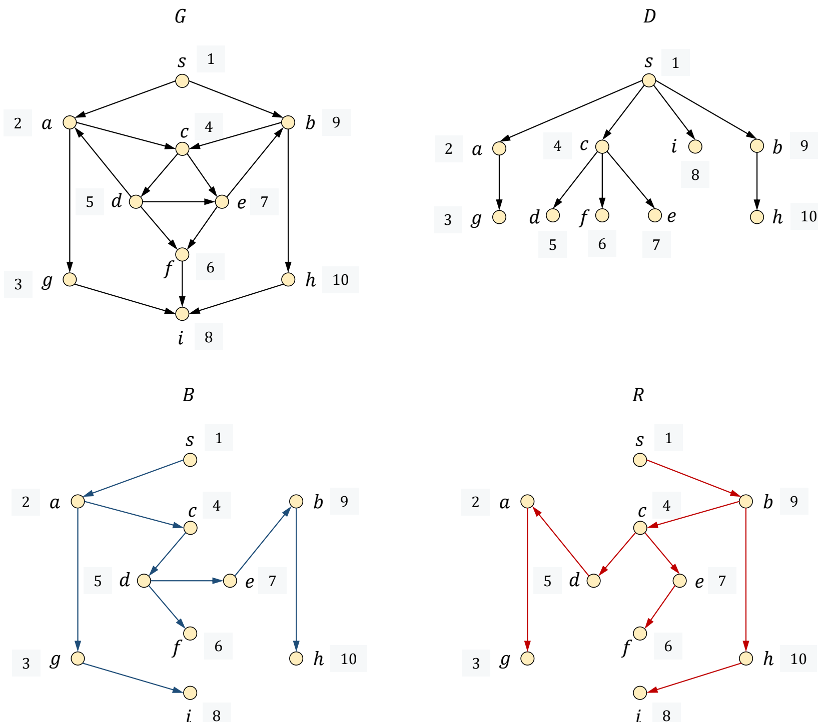


Figure 1 A flow graph G, its dominator tree D, and two strongly divergent spanning trees B and R. The numbers correspond to a preorder numbering of D that is a low-high order of G.

such path exists. The *dominator relation* in G is defined for the set of reachable vertices as follows. A vertex v is a *dominator* of a vertex w (v *dominates* w) if every path from s to w contains v; v is a *proper dominator* of w if v dominates w and $v \neq w$. The dominator relation in G can be represented by a tree rooted at s, the *dominator tree* D, such that v dominates w if and only if v is an ancestor of w in D. If $w \neq s$ is reachable, we denote by $d(w)$ the parent of w in D. Lengauer and Tarjan [34] presented an algorithm for computing dominators in $O(m\alpha(m,n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [44]. Subsequently, several linear-time algorithms were discovered [3, 10, 15, 16]. The dominator tree is a central tool in program optimization and code generation [12], and it has applications in other diverse areas including constraint programming [40], circuit testing [5], theoretical biology [2], memory profiling [36], the analysis of diffusion networks [28], and in connectivity problems [18, 19, 21, 20, 23, 29, 30, 31, 32].

A *low-high order* δ *of* G [25] is a preorder of the dominator tree D such for all reachable vertices $v \neq s$, $(d(v), v) \in E$ or there are two edges $(u, v) \in E$, $(w, v) \in E$ such that u and w are reachable, u is less than v ($u <_\delta v$), v is less than w ($v <_\delta w$), and w is not a descendant of v in D. See Figure 1. Every flow graph G has a low-high order, computable in linear-time [25]. Low-high orders provide a correctness certificate for dominator trees that is straightforward to verify [47]. By augmenting an algorithm that computes the dominator tree D of a flow graph G so that it also computes a low-high order of G, one obtains a *certifying algorithm* to compute D. (A *certifying algorithm* [37] outputs both the solution and a correctness certificate, with the property that it is straightforward to use the certificate to verify that the computed solution is correct.) Low-high orders also have applications in path-determination problems [46] and in fault-tolerant network design [6, 7, 26].

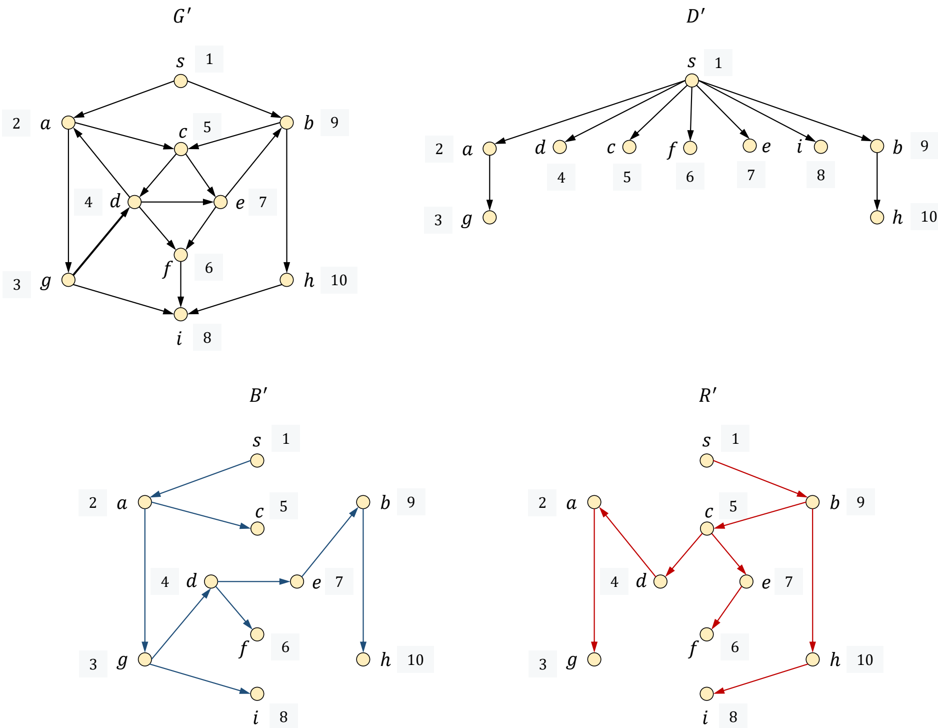


Figure 2 The flow graph of Figure 1 after the insertion of edge (g, d), and its updated dominator tree D' with a low-high order, and two strongly divergent spanning trees B' and R'.

A notion closely related to low-high orders is that of divergent spanning trees [25]. Let V_r be the set of reachable vertices, and let $G[V_r]$ be the flow graph with start vertex s that is induced by V_r. Two spanning trees B and R of $G[V_r]$, rooted at s, are *divergent* if for all v, the paths from s to v in B and R share only the dominators of v; B and R are *strongly divergent* if for every pair of vertices v and w, either the path in B from s to v and the path in R from s to w share only the common dominators of v and w, or the path in R from s to v and the path in B from s to w share only the common dominators of v and w. In order to simplify our notation, we will refer to B and R, with some abuse of terminology, as strongly divergent spanning trees of G. Every flow graph has a pair of strongly divergent spanning trees. Given a low-high order of G, it is straightforward to compute two strongly divergent spanning trees of G in $O(m)$ time [25]. Divergent spanning trees can be used in data structures that compute pairs of vertex-disjoint s-t paths in 2-vertex connected digraphs (for any two query vertices s and t) [18], in fast algorithms for approximating the smallest 2-vertex-connected spanning subgraph of a digraph [19], and in constructing sparse subgraphs of a given digraph that maintain certain connectivity requirements [21, 31, 32].

In this paper we consider how to update a low-high order of a flow graph through a sequence of edge insertions. See Figure 2. The difficulty in updating the dominator tree and a low-high order is due to the following facts. An affected vertex can be arbitrarily far from the inserted edge, and a single edge insertion may cause $O(n)$ parent changes in D. Furthermore, since a low-high order is a preorder of D, a single edge insertion may cause $O(n)$ changes in this order, even if there is only one vertex that is assigned a new parent in D after the insertion. More generally, we note that the hardness of dynamic algorithms on digraphs

has been recently supported also by conditional lower bounds [1]. Our first contribution is to show that we can maintain a low-high order of a flow graph G with n vertices through a sequence of edge insertions in $O(mn)$ total time, where m is the total number of edges after all insertions. Hence, we obtain a substantial improvement over the naive solution of recomputing a low-high order from scratch after each edge insertion, which takes $O(m^2)$ total time. Our result also implies the first incremental *certifying algorithms* [37] for computing dominators in $O(mn)$ total time, which answers an open question in [25]. We present two algorithms that achieve this bound, a simple algorithm based on sparsification and a more sophisticated algorithm. Both algorithms combine the incremental dominators algorithm of [22] with the linear-time computation of two divergent spanning trees from [25]. Our sophisticated algorithm also applies a slightly modified version of a static low-high algorithm from [25] on an auxiliary graph. We remark that the incremental dominators problem arises in various applications, such as incremental data flow analysis and compilation [11, 17, 41, 42], distributed authorization [38], and in incremental algorithms for maintaining 2-connectivity relations in directed graphs [23]. We present some applications of our result on incremental low-high order maintenance to incremental connectivity problems in Appendix A.

We assess the merits of our algorithm in practical scenarios by conducting a thorough experimental study with graphs taken from a variety of application areas. Although both the sparsification algorithm and the sophisticated algorithm have the same worst-case running time, our experimental results show that a carefully engineered implementation of the latter is by far superior in practice.

For lack of space, some proofs are omitted from this extended abstract. They are provided in the full version [24].

2 Preliminaries

Let $G = (V, E, s)$ be a flow graph with start vertex s, and let D be the dominator tree of G. A spanning tree T of G is a tree with root s that contains a path from s to v for all reachable vertices v. We refer to a spanning subgraph F of T as a spanning forest of G. Given a rooted tree T, we denote by $T(v)$ the subtree of T rooted at v (we also view $T(v)$ as the set of descendants of v). Let T be a tree rooted at s with vertex set $V_T \subseteq V$, and let $t(v)$ denote the parent of a vertex $v \in V_T$ in T. If v is an ancestor of w, $T[v, w]$ is the path in T from v to w. In particular, $D[s, v]$ consists of the vertices that dominate v. If v is a proper ancestor of w, $T(v, w]$ is the path to w from the child of v that is an ancestor of w. Tree T is *flat* if its root is the parent of every other vertex. Suppose now that the vertex set V_T of T consists of the vertices reachable from s. Tree T has the *parent property* if for all $(v, w) \in E$ with v and w reachable, v is a descendant of $t(w)$ in T. If T has the parent property and has a low-high order, then $T = D$ [25]. For any vertex $v \in V$, we denote by $C(v)$ the set of children of v in D. A *preorder* of T is a total order of the vertices of T such that, for every vertex v, the descendants of v are ordered consecutively, with v first. Let ζ be a preorder of D. Consider a vertex $v \neq s$. We say that ζ *is a low-high order for* v *in* G, if $(d(v), v) \in E$ or there are two edges $(u, v) \in E$, $(w, v) \in E$ such that $u <_\zeta v$ and $v <_\zeta w$, and w is not a descendant of v in D. Given a graph $G = (V, E)$ and a set of edges $S \subseteq V \times V$, we denote by $G \cup S$ the graph obtained by inserting into G the edges of S.

3 Incremental low-high order

In this section we describe two algorithms to maintain a low-high order of a digraph through a sequence of edge insertions. We first review some useful facts for updating a dominator

tree after an edge insertion [4, 22, 41]. Let (x, y) be the edge to be inserted. We consider the effect of this insertion when both x and y are reachable. Let G' be the flow graph that results from G after inserting (x, y). Similarly, if D is the dominator tree of G before the insertion, we let D' be the dominator tree of G'. Also, for any function f on V, we let f' be the function after the update. We say that vertex v is *affected* by the update if $d(v)$ (its parent in D) changes, i.e., $d'(v) \neq d(v)$. We let A denote the set of affected vertices. Note that we can have $D'[s, v] \neq D[s, v]$ even if v is not affected. We let $nca(x, y)$ denote the nearest common ancestor of x and y in the dominator tree D. We also denote by $depth(v)$ the depth of a reachable vertex v in D. There are affected vertices after the insertion of (x, y) if and only if $nca(x, y)$ is not a descendant of $d(y)$ [41]. A characterization of the affected vertices is provided by the following lemma, which is a refinement of a result in [4].

▶ **Lemma 1** ([22]). *Suppose x and y are reachable vertices in G. A vertex v is affected after the insertion of edge (x, y) if and only if $depth(nca(x, y)) < depth(d(v))$ and there is a path π in G from y to v such that $depth(d(v)) < depth(w)$ for all $w \in \pi$. If v is affected, then it becomes a child of $nca(x, y)$ in D', i.e., $d'(v) = nca(x, y)$.*

The algorithm (DBS) in [22] applies Lemma 1 to identify affected vertices by starting a search from y (if y is not affected, then no other vertex is). To do this search for affected vertices, it suffices to maintain the outgoing and incoming edges of each vertex. These sets are organized as singly linked lists, so that a new edge can be inserted in $O(1)$ time. The dominator tree D is represented by the parent function d. We also maintain the depth in D of each reachable vertex. We say that a vertex v is *scanned*, if the edges leaving v are examined during the search for affected vertices, and that it is *visited* if there is a scanned vertex u such that (u, v) is an edge in G. By Lemma 1, a visited vertex v is scanned if $depth(nca(x, y)) < depth(d(v))$.

▶ **Lemma 2** ([22]). *Let v be a scanned vertex. Then v is a descendant of an affected vertex in D.*

3.1 Sparsification Algorithm

In this algorithm we maintain, after each insertion, a subgraph $H = (V, E_H)$ of G with $O(n)$ edges that has the same dominator tree as G. Then, we can compute a low-high order δ of H in $O(|E_H|) = O(n)$ time. Note that by the definition of H, δ is also a valid low-high order of G. An edge insertion is processed by the routine SparseInsertEdge, shown below. Subgraph H is formed by the edges of two divergent spanning trees B and R of G. After the insertion of an edge (x, y), where both x and y are reachable, we form a graph H' by inserting into H a set of edges $Last(A)$ found during the search for the set of affected vertices A. Specifically, $Last(A)$ contains edge (x, y) and, for each affected vertex $v \neq y$, the last edge on a path π_{yv} that satisfies Lemma 1. Then, we set $H' = H \cup Last(A)$. Finally, we compute a low-high order and two divergent spanning trees of H', which are also valid for G'. We can show that this algorithm runs in $O(mn)$ total time.

3.2 Local Low-High Order Algorithm

Here we develop a more sophisticated and more practical algorithm that maintains a low-high order δ of a flow graph $G = (V, E, s)$ through a sequence of edge insertions. Our algorithm uses the incremental dominators algorithm of [22] to update the dominator tree D of G after each edge insertion. We describe a process to update δ based on the relation among vertices in D that are affected by the insertion. This enables us to identify a subset of vertices for

Algorithm 1: SparseInsertEdge(G, D, δ, B, R, e).

Input: Flow graph $G = (V, E, s)$, its dominator tree D, a low-high order δ of G, two divergent spanning trees B and R of G, and a new edge $e = (x, y)$.

Output: Flow graph $G' = (V, E \cup (x, y), s)$, its dominator tree D', a low-high order δ' of G', and two divergent spanning trees B' and R' of G'.

1 Insert e into G to obtain G'.
2 **if** *x is unreachable in G* **then return** (G', D, δ, B, R)
3 **else if** *y is unreachable in G* **then**
4 Compute the dominator tree D', two divergent spanning trees B' and R', and a low-high order δ' of G'.
5 **return** $(G', D', \delta', B', R')$
6 **end**
7 Let $H = B \cup R$.
8 Compute the updated dominator tree D' of G' and return a list A of the affected vertices, and a list $Last(A)$ of the last edge entering each $v \in A$ in a path satisfying Lemma 1.
9 Compute the subgraph $H' = H \cup Last(A)$ of G'.
10 Compute the dominator tree D', two divergent spanning trees B' and R', and a low-high order δ' of H'.
11 **return** $(G', D', \delta', B', R')$

which we can compute a "local" low-high order, that can be extended to a valid low-high order of G after the update. We show that such a "local" low-high order can be computed by a slightly modified version of an algorithm from [25]. We apply this algorithm on a sufficiently small flow graph that is defined by the affected vertices, and is constructed using the concept of derived edges [45].

3.2.1 Derived edges and derived flow graphs

Derived graphs, first defined in [45], reduce the problem of finding a low-high order to the case of a flat dominator tree [25]. By the parent property of D, if (v, w) is an edge of G, the parent $d(w)$ of w is an ancestor of v in D. Let (v, w) be an edge of G, with w not an ancestor of v in D. Then, the *derived edge* of (v, w) is the edge $(\overline{v}, w)$, where $\overline{v} = v$ if $v = d(w)$, $\overline{v}$ is the sibling of w that is an ancestor of v if $v \neq d(w)$. If w is an ancestor of v in D, then the derived edge of (v, w) is null. Note that a derived edge $(\overline{v}, w)$ may not be an original edge of G. For any vertex $w \in V$ such that $C(w) \neq \emptyset$, we define the *derived flow graph of* w, denoted by $G_w = (V_w, E_w, w)$, as the flow graph with start vertex w, vertex set $V_w = C(w) \cup \{w\}$, and edge set $E_w = \{(\overline{u}, v) \mid v \in V_w$ and $(\overline{u}, v)$ is the non-null derived edge of some edge in $E\}$. By definition, G_w has flat dominator tree, that is, w is the only proper dominator of any vertex $v \in V_w \setminus \{w\}$. We can compute a low-high order δ of G by computing a low-high order δ_w in each derived flow graph G_w. Given these low-high orders δ_w, we can compute a low-high order of G in $O(n)$ time by a depth-first traversal of D. During this traversal, we visit the children of each vertex w in their order in δ_w, and number the vertices from 1 to n as they are visited. The resulting preorder of D is low-high on G. Our incremental algorithm identifies, after each edge insertion, a specific derived flow graph G_w for which a low-high order δ_w needs to be updated. Then, it uses δ_w to update the low-high order of the whole flow graph G. Still, computing a low-high order of G_w can be too expensive to give us the

desired running time. Fortunately, we can overcome this obstacle by exploiting a relation among the vertices that are affected by the insertion, as specified below. This allows us to compute δ_w in a contracted version of G_w.

3.2.2 Affected vertices

Let (x, y) be the inserted vertex, where both x and y are reachable. Consider the execution of algorithm DBS [22] that updates the dominator tree by applying Lemma 1. Suppose vertex v is scanned, and let q be the nearest affected ancestor of v in D. Then, by Lemma 1, vertex q is a child of $nca(x, y)$ in D', i.e., $d'(q) = nca(x, y)$, and v remains a descendant of q in D'.

Our next lemma provides a key result about the relation of the affected vertices in D.

▶ **Lemma 3.** *All vertices that are affected by the insertion of (x, y) are descendants of a common child c of $nca(x, y)$.*

We shall apply Lemma 3 to construct a flow graph G_A for the affected vertices. Then, we shall use G_A to compute a "local" low-high order that we extend to a valid low-high order of G'.

3.2.3 Low-high order augmentation

Let δ be a low-high order of G, and let δ' be a preorder of the dominator tree D' of G'. We say that δ' *agrees with* δ if the following condition holds for any pair of siblings u, v in D that are not affected by the insertion of (x, y): $u <_{\delta'} v$ if and only if $u <_{\delta} v$. We can show that there is a low-high order δ' of G' that agrees with δ. Moreover, we have the following result:

▶ **Lemma 4.** *Let δ' be a preorder of D' that agrees with δ. Let v be a vertex that is not a child of $nca(x, y)$ and is not affected by the insertion of (x, y). Then δ' is a low-high order for v in G'.*

We can apply Lemmata 1 and 4 to show that in order to compute a low-high order of G', it suffices to compute a low-high order for the derived flow graph G'_z, where $z = nca(x, y)$. Still, the computation of a low-high order of G'_z is too expensive to give us the desired running time. Fortunately, as we show next, we can limit these computations for a contracted version of G'_z, defined by the affected vertices.

Let δ be a low-high order of G before the insertion of (x, y). Also, let $z = nca(x, y)$, and let δ_z be a corresponding low-high order of the derived flow graph G_z. That is, δ_z is the restriction of δ to z and its children in D. Consider the child c of z that, by Lemma 3, is an ancestor of all the affected vertices. Let α and β, respectively, be the predecessor and successor of c in δ_z. Note that α or β may be null. An *augmentation of* δ_z is an order δ'_z of $C'(z) \cup \{z\}$ that results from δ_z by inserting the affected vertices arbitrarily around c, that is, each affected vertex is placed in an arbitrary position between α and c or between c and β.

▶ **Lemma 5.** *Let $z = nca(x, y)$, and let δ_z be a low-high order of the derived flow graph G_z before the insertion of (x, y). Also, let δ'_z be an augmentation of δ_z, and let δ' be a preorder of D' that extends δ'_z. Then, for each child v of z in D, δ' is a low-high order for v in G'.*

3.2.4 Algorithm

Now we are ready to describe our incremental algorithm for maintaining a low-high order δ of G. For each vertex v that is not a leaf in D, we maintain a list of its children $C(v)$ in D, ordered by δ. Also, for each vertex $v \neq s$, we keep two variables $low(v)$ and $high(v)$.

Algorithm 2: LocalInsertEdge($G, D, \delta, mark, low, high, e$).

Input: Flow graph $G = (V, E, s)$, its dominator tree D, a low-high order δ of G, arrays *mark*, *low* and *high*, and a new edge $e = (x, y)$.

Output: Flow graph $G' = (V, E \cup (x, y), s)$, its dominator tree D', a low-high order δ' of G', and arrays $mark'$, low' and $high'$.

1 Insert e into G to obtain G'.
2 **if** *x is unreachable in G* **then return** $(G', D, \delta, mark, low, high)$
3 **else if** *y is unreachable in G* **then**
4 Compute the dominator tree D' and a low-high order δ' of G', together with the corresponding arrays $mark'$, low', and $high'$.
5 **return** $(G', D', \delta', mark', low', high')$
6 **end**
7 Compute the nearest common ancestor z of x and y in D.
8 Compute the updated dominator tree D' of G' and return a list A of the affected vertices.
9 **foreach** *vertex $v \in A$* **do** $mark'(y) \leftarrow false$
10 **if** $z = x$ **then** $mark'(y) \leftarrow true$
11 Compute a low-high order ζ of the derived affected flow graph G_A.
12 Compute the updated low-high order δ' of G' by ordering the vertices in $A \cup \{c\}$ according to ζ.
13 **foreach** *vertex $v \in A \cup \{c\}$* **do**
14 find edges (u, v) and (w, v) such that $u <_{\delta'} v <_{\delta'} w$ and $w \notin D'(v)$
15 set $low'(v) \leftarrow u$ and $high'(v) \leftarrow w$
16 **end**
17 **return** $(G', D', \delta', mark', low', high')$

Variable $low(v)$ stores an edge (u, v) such that $u \neq d(v)$ and $u <_\delta v$; $low(v) = null$ if no such edge exists. Similarly, $high(v)$ stores an edge (w, v) such that and $v <_\delta w$ and w is not a descendant of v in D; $high(v) = null$ if no such edge exists. These variables are useful in the applications that we mention in Appendix A. Finally, we mark each vertex v such that $(d(v), v) \in E$. For simplicity, we assume that the vertices of G are numbered from 1 to n, so we can store the above information in corresponding arrays *low*, *high*, and *mark*. Note that for a reachable vertex v, we can have $low(v) = null$ or $high(v) = null$ (or both) only if $mark(v) = true$. Before any edge insertion, all vertices are unmarked, and all entries in arrays *low* and *high* are null. We initialize the algorithm and the associated data structures by executing a linear-time algorithm to compute the dominator tree D of G [3, 10] and a linear-time algorithm to compute a low-high order δ of G [25]. So, the initialization takes $O(m + n)$ time for a digraph with n vertices and m edges.

Next, we describe the main routine, LocalInsertEdge, to handle an edge insertion. We let (x, y) be the inserted edge. Also, if x and y are reachable before the insertion, we let $z = nca(x, y)$.

From Lemmata 4 and 5 it follows that our main task now is to order the affected vertices according to a low-high order of D'. To do this, we use an auxiliary flow graph $G_A = (V_A, E_A, z)$, with start vertex z, which we refer to as the *derived affected flow graph*. Flow graph G_A is essentially a contracted version of the derived flow graph G'_z (i.e., the derived graph of z after the insertion) as we explain later. The vertices of the derived affected flow graph G_A are z, the affected vertices of G, their common ancestor c in D that is a child

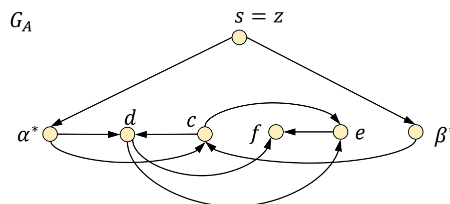


Figure 3 The derived affected flow graph G_A that corresponds to the flow graph of Figure 1 after the insertion of edge (g, d).

of z (from Lemma 3), and two auxiliary vertices α^* and β^*. Vertex α^* (resp., β^*) represents vertices in $C(z)$ with lower (resp., higher) order in δ than c. We include in G_A the edges (z, α^*) and (z, β^*). If c is marked then we include the edge (z, c) into G_A, otherwise we add the edges (α^*, c) and (β^*, c) into G_A. Also, for each edge (u, c) such that u is a descendant of an affected vertex v, we add in G_A the edge (v, c). Now we specify the edges that enter an affected vertex w in G_A. We consider each edge $(u, w) \in E$ entering w in G. We have the following cases:

(a) If u is a descendant of an affected vertex v, we add in G_A the edge (v, w).
(b) If u is a descendant of c but not a descendant of an affected vertex, then we add in G_A the edge (c, w).
(c) If $u \neq z$ is not a descendant of c, then we add the edge (α^*, w) if $u <_\delta c$, or the edge (β^*, w) if $c <_\delta u$.
(d) Finally, if $u = z$, then we add the edge (z, w). (In cases (c) and (d), $u = x$ and $w = y$.)

Recall that α (resp., β) is the sibling of c in D immediately before (resp., after) c in δ, if it exists. Then, we can obtain G_A from G'_z by contracting all vertices v with $v <_\delta c$ into $\alpha = \alpha^*$, and all vertices v with $c <_\delta v$ into $\beta = \beta^*$.

▶ **Lemma 6.** *The derived affected flow graph $G_A = (V_A, E_A, z)$ has flat dominator tree.*

Proof. We claim that for any two distinct vertices $v, w \in V_A \setminus z$, v does not dominate w. The lemma follows immediately from this claim. The claim is obvious for $w \in \{\alpha^*, \beta^*\}$, since G_A contains the edges (z, α^*) and (z, β^*). The same holds for $w = c$, since G_A contains the edge (z, c), or both the edges (α^*, c) and (β^*, c). Finally, suppose $w \in V_A \setminus \{z, \alpha^*, \beta^*\}$. Then, by the construction of G_A, vertex w is affected. By Lemma 3, $w \in D(c)$, which implies that there is a path in G from c to w that contains only vertices in $D(c)$. Hence, by construction, G_A contains a path from c to w that avoids α^* and β^*, so α^* and β^* do not dominate w. It remains to show that w is not dominated in G_A by c or another affected vertex v. Let (x, y) be the inserted edge. Without loss of generality, assume that $c <_\delta x$. Since w is affected, there is a path π in G from y to w that satisfies Lemma 1. Then π does not contain any vertex in $D[c, d(w)]$. Also, by the construction of G_A, π corresponds to a path π_A in G_A from β^* to y that avoids any vertex in $A \cap D[c, d(w)]$. Hence, w is not dominated by any vertex in $A \cap D[c, d(w)]$. It remains to show that w is not dominated by any affected vertex v in $A \setminus D[c, d(w)]$. Since both v and w are in $D(c)$ and v is not an ancestor of w in D, there is a path π' in G from c to w that contains only vertices in $D(c) \setminus \{v\}$. Then, by the construction of G_A, π' corresponds to a path π'_A in G_A from c to w that avoids v. Thus, v does not dominate w in G_A. ◀

▶ **Lemma 7.** *Let ν and μ, respectively, be the number of scanned vertices and their adjacent edges. Then, the derived affected flow graph G_A has $\nu + 4$ vertices, at most $\mu + 5$ edges, and can be constructed in $O(\nu + \mu)$ time.*

Proof. The bound on the number of vertices and edges in G_A follows from the definition of the derived affected flow graph. Next, we consider the construction time of G_A. Consider the edges entering the affected vertices. Let w be an affected vertex, and let $(u, w) \neq (x, y)$ be an edge of G'. Let q be nearest ancestor u in $C'(z)$. We distinguish two cases:

- u is not scanned. In this case, we argue that $q = c$. Indeed, it follows from the parent property of D and Lemma 3 that both u and w are descendants of c in D. Since u is not scanned, no ancestor of u in D is affected, so u remains a descendant of c in D'. Thus, $q = c$.
- u is scanned. Then, by Lemma 2, q is the nearest affected ancestor of u in D.

So we can construct the edges entering the affected vertices in G_A in two phases. In the first phase we traverse the descendants of each affected vertex q in D'. At each descendant u of q, we examine the edges leaving u. When we find an edge (u, w) with w affected, then we insert into G_A the edge (q, w). In the second phase we examine the edges entering each affected vertex w. When we find an edge (u, w) with u not visited during the first phase (i.e., u was not scanned during the update of D), we insert into G_A the edge (c, w). Note that during this construction we may insert the same edge multiple times, but this does not affect the correctness or running time of our overall algorithm. Since the descendants of an affected vertex are scanned, it follows that each phase runs in $O(\nu + \mu)$ time.

Finally, we need to consider the inserted edge (x, y). Let f be the nearest ancestor of x that is in $C(z)$. Since y is affected, $c \neq f$. Hence, we insert into G_A the edge (β^*, y) if $c <_\delta f$, and the edge (α^*, y) if $f <_\delta c$. Note that f is found during the computation of $z = nca(x, y)$, so this test takes constant time. ◀

Next, we order the vertices in $C'(z)$ according to a low-high order of ζ of G_A as follows. After computing G_A, we construct two divergent spanning trees B_A and R_A of G_A. For each vertex $v \neq z$, if (z, v) is an edge of G_A, we replace the parent of v in B_A and in R_A, denoted by $b_A(v)$ and $r_A(v)$, respectively, by z. We can compute a low-high order ζ of G_A by applying a slightly modified version of a linear-time algorithm of [25, Section 6.1] to compute a low-high order. Our modified version computes a low-high order ζ of G_A that is an augmentation of δ_z. To obtain such a low-high order, we need to assign to α^* the lowest number in ζ and to β^* the highest number in ζ. The algorithm works as follows. While G_A contains at least four vertices, we choose a vertex $v \notin \{\alpha^*, \beta^*\}$ whose in-degree in G_A exceeds its number of children in B_A plus its number of children in R_A and remove it from G_A. (From this choice of v we also have that $v \neq z$.) Then we compute recursively a low-high order for the resulting flow graph, and insert v in an appropriate location, defined by $b_A(v)$ and $r_A(v)$.

The correctness of algorithm LocalInsertEdge follows from Lemmata 4, 5 and 6. Also, by using Lemma 7, we can show that the total running time of the algorithm is bounded by $O(mn)$.

3.3 Representation of a low-high order

We consider two main options for representing a low-high order. The most straightforward is to maintain it as a preorder numbering of D, by assigning a preorder number from 1 to n to each vertex. Another option is to use a data structure for the dynamic list order problem [8, 14]. We experimented with various implementations of dynamic list order

Table 1 Real-world graphs with timestamped edges, from the Koblenz Network Collection [33].

Graph	nodes	reachable nodes	edges	avg. degree	type
temporalGraph	2029	1281	5517	2.72	Democratic National Council emails
opsahl-ucsocial	1899	1854	20296	10.69	UC Irvine messages
chess	7301	6312	60046	8.22	Chess games
munmun_digg_reply	30398	13471	85247	2.8	Digg replies
elec	7115	2316	103617	14.56	Wikipedia elections
slashdot-threads	51083	18851	130370	2.55	Slashdot threads

data structures, and in our experiments the best performance was achieved by a two-level numbering scheme that supports insertions, deletions and order queries in constant amortized time [8]. We remark that these operations suffice in all applications of our incremental algorithm that we mention in Appendix A.

Both of the above options suffices to have an implementation of the sparsification algorithm and of the local low-high order algorithm that run in total $O(mn)$ time. Since the sparsification algorithm computes the complete low-high order of a sparse subgraph of G after each update, there is no gain in using the more sophisticated numbering scheme that the dynamic list order data structure applies. For our local low-high order algorithm, however, the representation we choose is crucial for the practical performance of the algorithm. Specifically, using a dynamic list order data structure allows us to update the low-high order in amortized time proportional to the number of scanned vertices. The simple preorder numbering scheme, on the other hand, may need to renumber $O(n)$ vertices after a single update. Indeed, our preliminary experimental results confirmed that that the implementation that employs a dynamic list order data structure has superior performance.

3.4 Handling unreachable vertices

Now we provide some details on how our algorithms handle insertions of edges (x, y) when $x \in V_r$ and $y \notin V_r$, i.e., when only x is reachable from s before the insertion. In order to achieve $O(mn)$ total running time, we can simply recompute a low-high order from scratch after each such an insertion. This follows from the fact that there are at most $n - 1$ such insertions, and that we can recompute a low-high order in linear time when this type of an insertion occurs.

An alternative method, that performs much better in practice, is to compute the dominator tree and a low-high order for the vertices that were reachable from y but not from s before the insertion. Specifically, let Y be this set of vertices, and let $G[Y]$ be the flow graph with start vertex y that is induced by Y. Then, to handle the insertion of (x, y) we execute the following steps:

1. Compute the dominator tree D_Y of $G[Y]$ and a low-high order of it.
2. Link the dominator tree D of G with D_Y by making y a child of x in D, and merge appropriately the corresponding low-high orders.
3. Compute the set of edges E_Y from Y to V_r. Process each such edge as a regular insertion.

Note that after Step 2, D is the correct dominator tree for $G \setminus E_Y$ and we also have a valid low-high order of it. The insertion of the edges $(u, v) \in E_Y$ is handled as a regular insertion since both u and v are reachable from s after Step 2. In terms of running time, Steps 1 and 2 take $O(m)$ time. Also, since in Step 3 we have regular insertions, the total running time remains $O(mn)$.

Table 2 Real-world graphs used in the experiments, sorted by the file size of their largest strongly connected component (SCC). In our experiments we used both the largest SCC and the some of the 2-vertex-connected subgraphs (2VCSs), found inside the largest SCC.

Graph	Largest SCC			2VCSs			Type
	n	m	avg. degree	n	m	avg. degree	
rome99	3352	8855	2.64	2249	6467	2.88	road network [13]
twitter-higgs-retweet	13086	63537	4.86	1099	9290	8.45	twitter [35]
enron	8271	147353	17.82	4441	123527	27.82	enron mails [35]
web-NotreDame	48715	267647	5.49	1409	6856	4.87	web [35]
				1462	7279	4.98	
				1416	13226	9.34	
soc-Epinions1	32220	442768	13.74	17117	395183	23.09	trust network [35]
Amazon-302	241761	1131217	4.68	55414	241663	4.36	co-purchase [35]
WikiTalk	111878	1477665	13.21	49430	1254898	25.39	Wiki communications [35]
web-Stanford	150475	1576157	10.47	5179	129897	25.08	web [35]
				10893	162295	14.90	
web-Google	434818	3419124	7.86	77480	840829	10.85	web [35]
Amazon-601	395230	3301051	8.35	276049	2461072	8.92	co-purchase [35]
web-BerkStan	334857	4523232	13.51	1106	8206	7.42	web [35]
				4927	28142	5.71	
				12795	347465	27.16	
				29145	439148	15.07	

4 Empirical Analysis

We wrote our implementations in `C++`, using `g++ v.4.6.4` with full optimization (flag `-O3`) to compile the code. We report the running times on a GNU/Linux machine, with Ubuntu (12.04LTS): a Dell PowerEdge R715 server 64-bit NUMA machine with four AMD Opteron 6376 processors and 128GB of RAM memory. Each processor has 8 cores sharing a 16MB L3 cache, and each core has a 2MB private L2 cache and 2300MHz speed. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `getrusage` function, averaged over ten different runs. In Tables 1 and 2 we can see some statistics about the real-world graphs we used in our experimental evaluation. In all test instances we select the first vertex of the graph as the start vertex. (Choosing a random start vertex produces similar results.) Note that the graphs in Table 1 are not strongly connected so we also report the number of vertices that are reachable from the start vertex.

The graphs in Table 1 have timestamps that indicate the moment that each edge was inserted into the graph. Thus, in our experiments, the edges are inserted according to these timestamps. The number of edges that are actually inserted is controlled by a parameter $i \in [0, 1]$ as follows. Let m be the total number of edges in the graph. Then the flow graph initially has $m - i \cdot m$ edges, and $i \cdot m$ edges are inserted one at a time. The algorithms compute (in static mode) the dominator tree and a low-high order for the first $m - i \cdot m$ edges in the original graph file and then they run in incremental mode. Note that during the execution of the algorithms some vertices may be unreachable at first and become reachable after some insertions.

We use the graphs in Table 2 to create different types of inputs by extracting their largest strongly connected component and some large 2-vertex-connected subgraphs. (A 2-vertex-connected graph remains strongly connected after the deletion of any single vertex.) We use strongly connected graphs to guarantee that all vertices are reachable from any arbitrary start vertex. Also, the 2-vertex-connected graphs are interesting because they have flat dominator trees, so inserting their edges may cause the incremental algorithms to

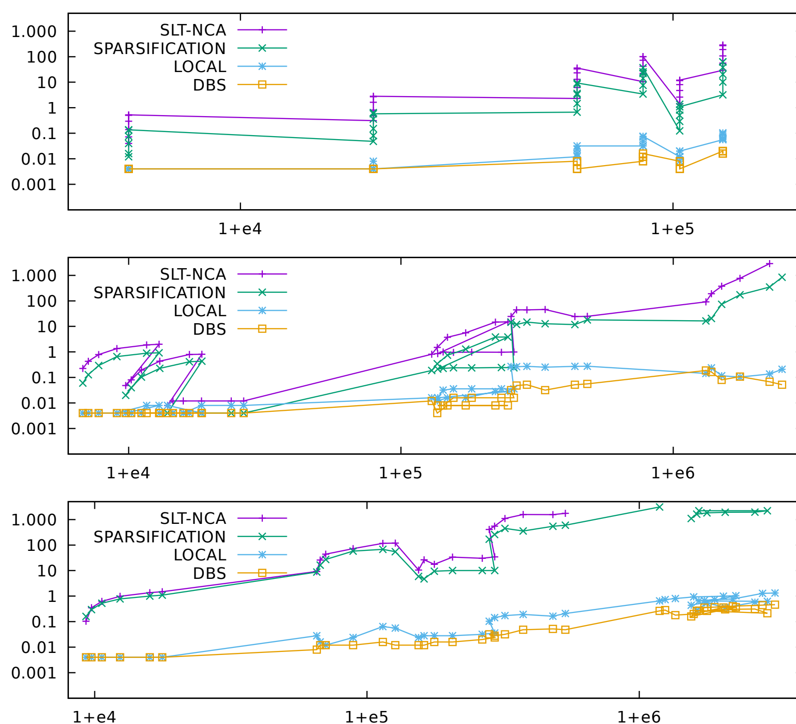


Figure 4 Incremental low-high order: timestamped graphs of Table 1 (top), random edge permutation of 2-vertex-connected graphs of Table 2 (middle), and random edge insertion in strongly connected graphs of Table 2 (bottom). Running times, in seconds, and number of edges both shown in logarithmic scale. For each input graph and each algorithm, we show the running times for inserting 5%, 10%, 20%, 40%, 80% and 100% of the edges.

perform a lot of work. We note that the graphs in Table 2 do not have timestamps, so we consider two different methods to produce dynamic graphs.

- **Random permutation:** We perform a random permutation of the edges and use the resulting order as timestamps.
- **Random insertions:** We insert $i \cdot m$ random edges in the original graph. The endpoints of each new edge are selected uniformly at random, and the edge is inserted if it is not a loop and is not already present in the current graph. Hence, the final graph has $m + i \cdot m$ edges.

We apply the first method to the 2-vertex-connected graphs, and the second method to the strongly connected graphs of Table 2. Note that in the case of random insertions the graph is strongly connected throughout the execution of the incremental algorithms. We do not apply the random insertions method to 2-vertex-connected graphs, since any edge insertion in such a graph has no effect on the dominator tree (so also the low-high order does not change).

We compare the performance of four algorithms. As a baseline, we use a static low-high order algorithm from [25] based on an efficient implementation of the Lengauer-Tarjan algorithm for computing dominators [34] from [27]. Our baseline algorithm, SLT-NCA, constructs, as intermediary, two divergent spanning trees. After each insertion of an edge (x, y), SLT-NCA tests if the insertion of (x, y) affects the dominator tree by computing the nearest common ancestor of x and y. If this is the case, then it recomputes a low-high order. The other two algorithms are the ones we presented in Section 3. For our sparsification

Table 3 Running times of the plot shown in Figure 4 (top): timestamped graphs of Table 1.

Graph	nodes	starting edges	final edges	SLT-NCA	SPARSIFICATION	LOCAL	DBS
temporalGraph05	2029	5241	5517	0.04	0.012	0.004	0.004
temporalGraph10	2029	4965	5517	0.072	0.016	0.004	0.004
temporalGraph20	2029	4414	5517	0.14	0.04	0.004	0.004
temporalGraph40	2029	3310	5517	0.296	0.076	0.004	0.004
temporalGraph80	2029	1103	5517	0.504	0.132	0.004	0.004
temporalGraph100	2029	0	5517	0.524	0.136	0.004	0.004
opsahl-ucsocial05	1899	19281	20296	0.312	0.048	0.004	0.004
opsahl-ucsocial10	1899	18266	20296	0.48	0.08	0.004	0.004
opsahl-ucsocial20	1899	16237	20296	0.812	0.152	0.004	0.004
opsahl-ucsocial40	1899	12178	20296	1.64	0.36	0.004	0.004
opsahl-ucsocial80	1899	4059	20296	2.68	0.64	0.008	0.004
opsahl-ucsocial100	1899	0	20296	2.796	0.576	0.004	0.004
chess05	7301	57044	60046	2.304	0.668	0.012	0.008
chess10	7301	54041	60046	6.14	1.456	0.016	0.008
chess20	7301	48037	60046	12.984	3.244	0.016	0.008
chess40	7301	36028	60046	23.28	3.64	0.024	0.004
chess80	7301	12009	60046	32.956	8.376	0.024	0.004
chess100	7301	0	60046	35.744	9.336	0.032	0.004
munmun_digg_reply05	30398	80985	85247	10.428	3.436	0.032	0.008
munmun_digg_reply10	30398	76722	85247	22.048	7.508	0.032	0.016
munmun_digg_reply20	30398	68198	85247	41.928	14.048	0.032	0.008
munmun_digg_reply40	30398	51148	85247	72.28	25.408	0.048	0.012
munmun_digg_reply80	30398	17049	85247	100.56	21.964	0.072	0.012
munmun_digg_reply100	30398	0	85247	100.98	37.156	0.076	0.016
elec05	7115	98436	103617	1.408	0.124	0.012	0.008
elec10	7115	93255	103617	2.62	0.292	0.012	0.004
elec20	7115	82894	103617	4.732	0.508	0.008	0.008
elec40	7115	62170	103617	8.08	0.812	0.012	0.004
elec80	7115	20723	103617	11.52	1.364	0.02	0.008
elec100	7115	0	103617	12.068	1.096	0.02	0.004
slashdot-threads05	51083	123852	130370	29.412	3.2	0.056	0.02
slashdot-threads10	51083	117333	130370	56.112	10.184	0.06	0.02
slashdot-threads20	51083	104296	130370	106.772	20.144	0.06	0.02
slashdot-threads40	51083	78222	130370	189.712	37.996	0.064	0.016
slashdot-threads80	51083	26074	130370	287.912	62.66	0.092	0.02
slashdot-threads100	51083	0	130370	270.356	62.532	0.104	0.016

algorithm of Section 3.1, denoted as SPARSIFICATION, we extend the incremental dominators algorithm DBS of [22] with the computation of two divergent spanning trees and a low-high order. Algorithm SPARSIFICATION applies these computations on a sparse subgraph of the input digraph that maintains the same dominators. Finally, we tested an implementation of our more efficient algorithm of Section 3.2, denoted as LOCAL, that updates the low-high order by computing a local low-high order of an auxiliary graph. We include also the original DBS of [22] in the experiments, to provide a more complete picture of the effectiveness of these approaches.

We compared the above incremental low-high order algorithms in three different field tests, as mentioned above. The first one, shown in Figure 4 (top) and Table 3, compares the running times of the algorithms against the dataset detailed in Table 1, i.e. the timestamped graphs. The algorithms are well distinguished: our more efficient algorithm, LOCAL, performs very well. Indeed, its running time is very close to DBS that only updates the dominator tree. Algorithm SPARSIFICATION is not competitive with LOCAL, despite the fact that it exhibits a substantial improvement over our baseline algorithm SLT-NCA.

The second experiment, shown in Figure 4 (middle) and Table 4, deals with the random permutations of the edges of 2-vertex-connected graphs. (Refer to Table 2.) As with the timestamped graphs, during the execution of the algorithms some vertices may be unreachable at first, but here all vertices become reachable in the end. Also, at the end of all insertions, the final graph has flat dominator tree. Here we can see that, as before, the algorithms are still distinguished, but in this case the two couples SPARSIFICATION and SLT-NCA, and LOCAL and DBS, are closer.

Table 4 Running times of the plot shown in Figure 4 (middle): random edge permutation of 2-vertex-connected graphs of Table 2.

Graph	nodes	starting edges	final edges	SLT-NCA	SPARSIFICATION	LOCAL	DBS
rome05	2249	6467	6790	0.224	0.06	0.004	0.004
rome10	2249	6467	7114	0.428	0.128	0.004	0.004
rome20	2249	6467	7760	0.784	0.292	0.004	0.004
rome40	2249	6467	9054	1.348	0.656	0.004	0.004
rome80	2249	6467	11641	1.868	0.896	0.008	0.004
rome100	2249	6467	12934	1.992	0.92	0.008	0.004
twitter05	1099	9290	9755	0.048	0.02	0.004	0.004
twitter10	1099	9290	10219	0.08	0.04	0.004	0.004
twitter20	1099	9290	11148	0.196	0.104	0.004	0.004
twitter40	1099	9290	13006	0.432	0.228	0.004	0.004
twitter80	1099	9290	16722	0.796	0.412	0.004	0.004
twitter100	1099	9290	18580	0.812	0.436	0.004	0.004
NotreDame05	1416	13226	13887	0.008	0.004	0.008	0.004
NotreDame10	1416	13226	14549	0.012	0.004	0.004	0.004
NotreDame20	1416	13226	15871	0.012	0.004	0.004	0.004
NotreDame40	1416	13226	18516	0.012	0.004	0.008	0.004
NotreDame80	1416	13226	23807	0.012	0.004	0.008	0.004
NotreDame100	1416	13226	26452	0.012	0.004	0.008	0.004
enron05	4441	123527	129703	0.808	0.188	0.016	0.012
enron10	4441	123527	135880	1.504	0.344	0.012	0.004
enron20	4441	123527	148232	3.744	0.748	0.016	0.008
enron40	4441	123527	172938	5.584	1.28	0.016	0.008
enron80	4441	123527	222349	14.744	3.836	0.028	0.008
enron100	4441	123527	247054	15.076	3.828	0.028	0.008
webStanford05	5179	129897	136392	0.856	0.212	0.016	0.008
webStanford10	5179	129897	142887	0.992	0.228	0.032	0.008
webStanford20	5179	129897	155876	0.964	0.24	0.036	0.016
webStanford40	5179	129897	181856	0.98	0.236	0.036	0.016
webStanford80	5179	129897	233815	0.968	0.244	0.036	0.016
webStanford100	5179	129897	259794	0.988	0.24	0.036	0.016
Amazon05	55414	241663	253746	24.528	14.592	0.268	0.032
Amazon10	55414	241663	265829	44.392	11.88	0.264	0.048
Amazon20	55414	241663	289996	44.356	14.704	0.272	0.052
Amazon40	55414	241663	338328	45.792	12.628	0.252	0.032
Amazon80	55414	241663	434993	24.22	11.82	0.272	0.052
Amazon100	55414	241663	483326	24.856	18.096	0.276	0.056
WikiTalk05	49430	1254898	1317643	91.808	16.344	0.144	0.188
WikiTalk10	49430	1254898	1380388	190.616	20.42	0.236	0.164
WikiTalk20	49430	1254898	1505878	374.292	73.252	0.116	0.08
WikiTalk40	49430	1254898	1756857	766.28	172.064	0.104	0.108
WikiTalk80	49430	1254898	2258816	2868.28	349.632	0.136	0.068
WikiTalk100	49430	1254898	2509796	> 3600	837.728	0.208	0.052

The last experiment, detailed in Figure 4 (bottom) and Table 5, concerns the random edge insertion in strongly connected graphs of Table 2. The ranking of the algorithms does not change, as we can see in Figure 4 (bottom), but the difference is bigger: we note a bigger gap of more than two orders of magnitude, in particular, between LOCAL and the couple SLT-NCA and SPARSIFICATION.

From all the above experimental results, it is evident that a careful implementation of our efficient algorithm LOCAL has excellent performance in practice. Indeed, its running time is very close to the running time of an efficient incremental algorithm for updating the dominator tree.

Table 5 Running times of the plot shown in Figure 4 (bottom): random edge insertion in strongly connected graphs of Table 2.

Graph	nodes	starting edges	final edges	SLT-NCA	SPARSIFICATION	LOCAL	DBS
rome05	3352	8855	9298	0.104	0.16	0.004	0.004
rome10	3352	8855	9741	0.352	0.296	0.004	0.004
rome20	3352	8855	10626	0.624	0.528	0.004	0.004
rome40	3352	8855	12397	0.98	0.78	0.004	0.004
rome80	3352	8855	15939	1.372	1	0.004	0.004
rome100	3352	8855	17710	1.48	1.088	0.004	0.004
twitter05	13086	63537	65444	9.252	8.74	0.028	0.008
twitter10	13086	63537	67344	25.716	16.452	0.016	0.012
twitter20	13086	63537	70544	44.148	27.124	0.012	0.012
twitter40	13086	63537	88952	72.732	57.828	0.024	0.012
twitter80	13086	63537	114367	116.452	68.1	0.064	0.016
twitter100	13086	63537	127074	119.152	54.624	0.056	0.012
enron05	8271	147353	154721	10.52	5.928	0.024	0.012
enron10	8271	147353	162088	26.512	4.712	0.028	0.012
enron20	8271	147353	176824	17.652	9.4	0.028	0.016
enron40	8271	147353	206294	33.724	10.044	0.028	0.016
enron80	8271	147353	265235	30.34	10.02	0.032	0.02
enron100	8271	147353	294706	34.496	10.012	0.036	0.024
NotreDame05	48715	267647	281029	409.072	169.168	0.104	0.032
NotreDame10	48715	267647	294412	550.444	259.42	0.144	0.028
NotreDame20	48715	267647	321176	1093.96	447.44	0.172	0.032
NotreDame40	48715	267647	374706	1575.83	356.588	0.192	0.048
NotreDame80	48715	267647	481765	1563.68	544.748	0.164	0.052
NotreDame100	48715	267647	535294	1753.4	597.776	0.208	0.048
Amazon05	241761	1131217	1187778	> 3600	3098.83	0.652	0.264
Amazon10	241761	1131217	1244339	> 3600	> 3600	0.732	0.284
Amazon20	241761	1131217	1357460	> 3600	> 3600	0.804	0.18
Amazon40	241761	1131217	1583704	> 3600	> 3600	0.936	0.2
Amazon80	241761	1131217	2036191	> 3600	> 3600	0.992	0.36
Amazon100	241761	1131217	2262434	> 3600	> 3600	1.032	0.368
WikiTalk05	111878	1477665	1551548	> 3600	1096.12	0.44	0.16
WikiTalk10	111878	1477665	1625432	> 3600	1619.04	0.28	0.264
WikiTalk20	111878	1477665	1773198	> 3600	1831.12	0.544	0.264
WikiTalk40	111878	1477665	2068731	> 3600	1932.21	0.36	0.304
WikiTalk80	111878	1477665	2659797	> 3600	1947	0.576	0.312
WikiTalk100	111878	1477665	2955330	> 3600	2207.42	0.62	0.212
webStanford05	150475	1576157	1654965	> 3600	2208.26	0.648	0.268
webStanford10	150475	1576157	1733773	> 3600	> 3600	0.676	0.356
webStanford20	150475	1576157	1891388	> 3600	> 3600	0.732	0.372
webStanford40	150475	1576157	2206620	> 3600	> 3600	0.768	0.404
webStanford80	150475	1576157	2837083	> 3600	> 3600	1.288	0.444
webStanford100	150475	1576157	3152314	> 3600	> 3600	1.324	0.464

References

1. A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. 55th IEEE Symposium on Foundations of Computer Science, FOCS*, pages 434–443, 2014. doi:10.1109/FOCS.2014.53.
2. S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004.
3. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.
4. S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical Report 96-3, Department of Computer Science, University of Copenhagen, 1996.
5. M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, March 2001.
6. S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability for directed graphs. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 528–543. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48653-5_35.
7. S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability subgraph: Generic and optimal. In *Proc. 48th ACM Symp. on Theory of Computing*, pages 509–518, 2016. doi:10.1145/2897518.2897648.
8. M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 152–164, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45749-6_17.
9. M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms*, 12(2):14:1–14:22, December 2015. doi:10.1145/2756553.
10. A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.
11. S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi-dynamic problems on digraphs. *Theor. Comput. Sci.*, 203:69–90, August 1998.
12. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. doi:10.1145/115372.115320.
13. C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. 2007. URL: http://www.diag.uniroma1.it/challenge9/.
14. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 365–372, 1987.
15. W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013. doi:http://dx.doi.org/10.1016/j.jda.2013.10.003.
16. H. N. Gabow. The minset-poset approach to representations of graph connectivity. *ACM Transactions on Algorithms*, 12(2):24:1–24:73, February 2016. doi:10.1145/2764909.
17. K. Gargi. A sparse algorithm for predicated global value numbering. *SIGPLAN Not.*, 37(5):45–56, May 2002. doi:10.1145/543552.512536.
18. L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint s-t paths in digraphs. In *Proc. 37th Int'l. Coll. on Automata, Languages, and Programming*, pages 738–749, 2010.

19 L. Georgiadis. Approximating the smallest 2-vertex connected spanning subgraph of a directed graph. In *Proc. 19th European Symposium on Algorithms*, pages 13–24, 2011.

20 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 605–616, 2015. doi:10.1007/978-3-662-47672-7_49.

21 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2edgee connectivity in directed graphs. *ACM Trans. Algorithms*, 13(1):9:1–9:24, October 2016. doi:10.1145/2968448.

22 L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *Proc. 20th European Symposium on Algorithms*, pages 491–502, 2012. Full version: *CoRR*, abs/1604.02711.

23 L. Georgiadis, G. F. Italiano, and N. Parotsidis. Incremental 2-edge-connectivity in directed graphs. In *Proc. 43rd Int'l. Coll. on Automata, Languages, and Programming*, pages 49:1–49:15, 2016. doi:10.4230/LIPIcs.ICALP.2016.49.

24 L. Georgiadis, A. Karanasiou, G. Konstantinos, and L. Laura. On low-high orders of directed graphs: Incremental algorithms and applications. *CoRR*, abs/1608.06462, 2016. URL: http://arxiv.org/abs/1608.06462.

25 L. Georgiadis and R. E. Tarjan. Dominator tree certification and divergent spanning trees. *ACM Transactions on Algorithms*, 12(1):11:1–11:42, November 2015. doi:10.1145/2764913.

26 L. Georgiadis and R. E. Tarjan. Addendum to "Dominator tree certification and divergent spanning trees". *ACM Transactions on Algorithms*, 12(4):56:1–56:3, August 2016. doi:10.1145/2928271.

27 L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications (JGAA)*, 10(1):69–94, 2006.

28 M. Gomez-Rodriguez and B. Schölkopf. Influence maximization in continuous time diffusion networks. In *29th International Conference on Machine Learning (ICML)*, 2012.

29 M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 713–724, 2015. doi:10.1007/978-3-662-47672-7_58.

30 G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012. doi:10.1016/j.tcs.2011.11.011.

31 R. Jaberi. Computing the 2-blocks of directed graphs. *RAIRO-Theor. Inf. Appl.*, 49(2):93–119, 2015. doi:10.1051/ita/2015001.

32 R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164–172, 2016. doi:10.1016/j.dam.2015.10.001.

33 J. Kunegis. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW'13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350, 2013.

34 T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.

35 J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014. URL: http://snap.stanford.edu/data.

36 E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD'10, pages 115–124, 2010.

37 R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

38 M. Mowbray and A. Lain. Dominator-tree analysis for distributed authorization. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS'08, pages 101–112, New York, NY, USA, 2008. ACM. `doi:10.1145/1375696.1375709`.

39 H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

40 L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proc. 8th International Conference on Practical Aspects of Declarative Languages*, pages 73–87, 2006.

41 G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proc. 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 287–296, 1994.

42 V. C. Sreedhar, G. R. Gao, and Y. Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19:239–252, 1997.

43 R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

44 R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

45 R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

46 T. Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theoretical Computer Science*, 465:35–48, 2012. `doi:10.1016/j.tcs.2012.09.025`.

47 J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Proc. 2nd International Conference on Certified Programs and Proofs*, pages 27–42. Springer, 2012. `doi:10.1007/978-3-642-35308-6_6`.

A Applications of incremental low-high orders

In this section we show how our result on incremental low-high order maintenance implies the following incremental algorithms that also run in $O(mn)$ total time for a sequence of m edge insertions.

- First, we give an algorithm that maintains, after each edge insertion, two strongly divergent spanning trees of G, and answers the following queries in constant time: (i) For any two query vertices v and w, find a path π_{sv} from s to v and a path π_{sw} from s to w, such that π_{sv} and π_{sw} share only the common dominators of v and w. We can output these paths in $O(|\pi_{sv}| + |\pi_{sw}|)$ time. (ii) For any two query vertices v and w, find a path π_{sv} from s to v that avoids w, if such a path exists. We can output this path in $O(|\pi_{sv}|)$ time.
- Then we provide an algorithm for an incremental version of the fault-tolerant reachability problem [6, 7]. We maintain a flow graph $G = (V, E, s)$ with n vertices through a sequence of m edge insertions, so that we can answer the following query in $O(n)$ time. Given a spanning forest $F = (V, E_F)$ of G rooted at s, find a set of edges $E' \subseteq E \setminus E_F$ of minimum cardinality, such that the subgraph $G' = (V, E_F \cup E', s)$ of G has the same dominators as G.
- Finally, given a digraph G, we consider how to maintain incrementally a spanning subgraph of G with $O(n)$ edges that preserves the 2-edge-connectivity relations in G.

A.1 Strongly divergent spanning trees and path queries

We can use the arrays *mark*, *low*, and *high* to maintain a pair of strongly divergent spanning trees, B and R, of G after each update. Recall that B and R are *strongly divergent* if for every pair of vertices v and w, we have $B[s,v] \cap R[s,w] = D[s,v] \cap D[s,w]$ or $R[s,v] \cap B[s,w] = D[s,v] \cap D[s,w]$. Moreover, we can construct B and R so that they are also edge-disjoint except for the bridges of G. A *bridge* of G is an edge (u,v) that is contained in every path from s to v. Let $b(v)$ (resp., $r(v)$) denote the parent of a vertex v in B (resp., R). To update B and R after the insertion of an edge (x,y), we only need to update $b(v)$ and $r(v)$ for the affected vertices v, and possibly for their common ancestor c that is a child of $z = nca(x,y)$ from Lemma 3. We can update $b(v)$ and $r(v)$ of each vertex $v \in A \cup \{c\}$ as follows: set $b(v) \leftarrow d(v)$ if $low(v) = null$, $b(v) \leftarrow low(v)$ otherwise; set $r(v) \leftarrow d(v)$ if $high(v) = null$, $r(v) \leftarrow high(v)$ otherwise. If the insertion of (x,y) does not affect y, then $A = \emptyset$ but we may still need to update $b(y)$ and $r(y)$ if $x \notin D(y)$ in order to make B and R maximally edge-disjoint. Note that in this case $z = d(y)$, so we only need to check if both $low(y)$ and $high(y)$ are null. If they are, then we set $low(y) \leftarrow x$ if $x <_\delta y$, and set $high(y) \leftarrow x$ otherwise. Then, we can update $b(y)$ and $r(y)$ as above.

Now consider a query that, given two vertices v and w, asks for two maximally vertex-disjoint paths, π_{sv} and π_{sw}, from s to v and from s to w, respectively. Such queries were used in [46] to give a linear-time algorithm for the 2-disjoint paths problem on a directed acyclic graph. If $v <_\delta w$, then we select $\pi_{sv} \leftarrow B[s,v]$ and $\pi_{sw} \leftarrow R[s,w]$; otherwise, we select $\pi_{sv} \leftarrow R[s,v]$ and $\pi_{sw} \leftarrow B[s,w]$. Therefore, we can find such paths in constant time, and output them in $O(|\pi_{sv}| + |\pi_{sw}|)$ time. Similarly, for any two query vertices v and w, we can report a path π_{sv} from s to v that avoids w. Such a path exists if and only if w does not dominate v, which we can test in constant time using the ancestor-descendant relation in D [43]. If w does not dominate v, then we select $\pi_{sv} \leftarrow B[s,v]$ if $v <_\delta w$, and select $\pi_{sv} \leftarrow R[s,v]$ if $w <_\delta v$.

A.2 Fault tolerant reachability

Baswana et al. [6] study the following reachability problem. We are given a flow graph $G = (V,E,s)$ and a spanning tree $T = (V,E_T)$ rooted at s. We call a set of edges E' *valid* if the subgraph $G' = (V, E_T \cup E', s)$ of G has the same dominators as G. The goal is to find a valid set of minimum cardinality. As shown in [26], we can compute a minimum-size valid set in $O(m)$ time, given the dominator tree D and a low-high order of δ of it. We can combine the above construction with our incremental low-high algorithm to solve the incremental version of the fault tolerant reachability problem, where G is modified by edge insertions and we wish to compute efficiently a valid set for any query spanning tree T. Let $t(v)$ be the parent of v in T. Our algorithm maintains, after each edge insertion, a low-high order δ of G, together with the *mark*, *low*, and *high* arrays. Given a query spanning tree $T = (V,E_T)$, we can compute a valid set of minimum cardinality E' as follows. For each vertex $v \neq s$, we apply the appropriate one of the following cases: (a) If $t(v) = d(v)$ then we do not insert into E' any edge entering v. (b) If $t(v) \neq d(v)$ and v is marked then we insert $(d(v),v)$ into E'. (c) If v is not marked then we consider the following subcases: If $t(v) >_\delta v$, then we insert into E' the edge (x,v) with $x = low(v)$. Otherwise, if $t(v) <_\delta v$, then we insert into E' the edge (x,v) with $x = high(v)$. Hence, can update the minimum valid set in $O(mn)$ total time.

We note that the above construction can be easily generalized for the case where T is forest, i.e., when E_T is a subset of the edges of some spanning tree of G. In this case, $t(v)$ can be null for some vertices $v \neq s$. To answer a query for such a T, we apply the previous

construction with the following modification when $t(v)$ is null. If v is marked then we insert $(d(v), v)$ into E', as in case (b). Otherwise, we insert both edges entering v from $low(v)$ and $high(v)$. In particular, when $E_T = \emptyset$, we compute a subgraph $G' = (V, E', s)$ of G with minimum number of edges that has the same dominators as G. This corresponds to the case $k = 1$ in [7].

A.3 Sparse certificate for 2-edge-connectivity

Let $G = (V, E)$ be a strongly connected digraph. We say that vertices $u, v \in V$ are 2*-edge-connected* if there are two edge-disjoint directed paths from u to v and two edge-disjoint directed paths from v to u. (A path from u to v and a path from v to u need not be edge-disjoint.) A 2*-edge-connected block* of a digraph $G = (V, E)$ is defined as a maximal subset $B \subseteq V$ such that every two vertices in B are 2-edge-connected. If G is not strongly connected, then its 2-edge-connected blocks are the 2-edge-connected blocks of each strongly connected component of G. A *sparse certificate* for the 2-edge-connected blocks of G is a spanning subgraph $C(G)$ of G that has $O(n)$ edges and maintains the same 2-edge-connected blocks as G. Sparse certificates of this kind allow us to speed up computations, such as finding the actual edge-disjoint paths that connect a pair of vertices (see, e.g., [39]). The 2-edge-connected blocks and a corresponding sparse certificate can be computed in $O(m + n)$ time [21]. An incremental algorithm for maintaining the 2-edge-connected blocks is presented in [23]. This algorithm maintains the dominator tree of G, with respect to an arbitrary start vertex s, and of its reversal G^R, together with the auxiliary components of G and G^R, defined next.

Recall that an edge (u, v) is a *bridge* of a flow graph G with start vertex s if all paths from s to v include (u, v). After deleting from the dominator tree D the bridges of G, we obtain the *bridge decomposition* of D into a forest $\mathcal{D}$. For each root r of a tree in the bridge decomposition $\mathcal{D}$ we define the *auxiliary graph* $G_r = (V_r, E_r)$ *of* r as follows. The vertex set V_r of G_r consists of all the vertices in D_r. The edge set E_r contains all the edges of G among the vertices of V_r, referred to as *ordinary* edges, and a set of *auxiliary* edges, which are obtained by contracting vertices in $V \setminus V_r$, as follows. Let v be a vertex in V_r that has a child w in $V \setminus V_r$. Note that w is a root in the bridge decomposition $\mathcal{D}$ of D. For each such child w of v, we contract w and all its descendants in D into v. The *auxiliary components* of G are the strongly connected components of each auxiliary graph G_r.

We sketch how to extend the incremental algorithm of [23] so that it also maintains a sparse certificate $C(G)$ for the 2-edge-connected components of G, in $O(mn)$ total time. It suffices to maintain the auxiliary components in G and G^R, and two maximally edge-disjoint divergent spanning trees for each of G and G^R. We can maintain these divergent spanning trees as described in Section A.1. To identify the auxiliary components, the algorithm of [23] uses, for each auxiliary graph, an incremental algorithm for maintaining strongly connected components [9]. It is easy to extend this algorithm so that it also computes $O(n)$ edges that define these strongly connected components. The union of these edges and of the edges in the divergent spanning trees are the edges of $C(G)$.

Jdrasil: A Modular Library for Computing Tree Decompositions

Max Bannach[1], Sebastian Berndt[2], and Thorsten Ehlers[3]

1 **Institute for Theoretical Computer Science, Universität zu Lübeck, Lübeck, Germany**
`bannach@tcs.uni-luebeck.de`
2 **Institute for Theoretical Computer Science, Universität zu Lübeck, Lübeck, Germany**
`berndt@tcs.uni-luebeck.de`
3 **Department of Computer Science, Kiel University, Kiel, Germany**
`the@informatik.uni-kiel.de`

Abstract

While the theoretical aspects concerning the computation of tree width – one of the most important graph parameters – are well understood, it is not clear how it can be computed *practically*. We present the open source Java library Jdrasil that implements several different state of the art algorithms for this task. By experimentally comparing these algorithms, we show that the default choices made in Jdrasil lead to an competitive implementation (it took the third place in the first PACE challenge) while also being easy to use and easy to extend.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases tree width, algorithmic library, experimental evaluation

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.28

1 Introduction

The concept of the *tree width* of a graph – the similarity of the graph to a tree – has seen an enormous amount of research in the last few years due to its theoretical and pratical importance. *Google Scholar*[1] lists more than 6.000 papers concerning this subject written in the last five years and more than 16.000 papers in total. More than half of the papers in the proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC 2015) mention this important graph notion [25]. Tree width as a measure of the complexity of a graph has shown to be helpful in a wide range of applications ranging from the analysis of genome structure (e. g. [32]) to the learning of probablistic network from a given dataset (e. g. [27]). It has also been shown to be very useful for the theoretical investigation of the computational complexity of several graph problems, as many problems that are intractable (i. e. NP-hard) become efficiently solvable on graphs with bounded tree width. Due to this fact, tree width plays a major part in the development of fixed-parameter algorithms in the field of parameterized complexity.

A wide range of algorithms is known to compute tree decompositions, ranging from experimental heuristics over to exact exponential-time algorithms. However, they usually suffer from at least one of the following problems:

[1] `https://scholar.google.com`

© Max Bannach, Sebastian Berndt, and Thorsten Ehlers;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 28; pp. 28:1–28:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1. The running time of the algorithm is too high even for medium-sized instances (e. g. graphs with $n \approx 100$ vertices) that arise in typical applications;
2. The value of the computed solution may be arbitrarily bad compared to the value of an optimal solution;
3. The algorithm itself may be quite complicated, which prevents an useful implementation of the algorithm.

Due to this problems, the first Parameterized Algorithms and Computational Experiments (PACE) challenge [28, 21] decided to choose the fast computation of tree decompositions as one of its tracks. The organizers wrote[2]:

> *The ambition of this track is to turn tree width, a concept that has been tremendously successful in theoretical work, into a practically useful tool. Many algorithms in parameterized complexity rely on the existence of tree decompositions of small width, and yet in practice we don't have a good understanding for how to actually compute such a decomposition. This has to change.*

This lack of practicality hinders research in theoretical and in practical investigations:

Case Example A: A researcher in bioinformatics has used very sophisticated algorithms to learn the causal graph describing the behaviour of the norovirus. This graphs consists of roughly 600 vertices. To speed up the following combinatorically complex algorithms, she would like to have an optimal tree decomposition of this graph. Unfortunately, the graph is to big to find an optimal decomposition with simple algorithms, and she tries, unsuccessfully, to find implementation of the more advanced algorithms on the internet.

Case Example B: A professor in parameterized complexity has developed a new algorithm that solves the graph 3-coloring problem using tree decompositions. He is aware of the classic algorithm by Arnborg and Proskurowski [2], but believes that his algorithm is practically more feasible. In order to evaluate his claim, he wishes to give out a bachelor thesis to a promising student, who should implement and compare both algorithms. However, the first step of these algorithms — computing a tree decomposition -– almost busts the scale of bachelor thesis, and the new algorithm ends up added to the fast growing list of never implemented algorithms.

Case Example C: A Ph. D. student has developed a new parallel algorithm to compute optimal tree decompositions, and she now wishes to implement and test this algorithm. However, before she can actually start with the "real" implementation, she has to take care of a lot of other things: data structures for graphs, and more involved ones for tree decompositions. She also has to implement all kinds of difficult graph parameters used by her new algorithm. Finally, in order to be competitive, she also has to implement all the known pre- and post-processing algorithms for tree width. Before she can actually start, a long time has passed.

1.1 Our Contributions

In this work, we aim to provide solutions for these use cases, and to broaden the understanding of the behaviour of several different approaches for the computation of tree width.

1. We provide the Java library Jdrasil that implements several different tree width algorithms (exact and heuristic). This library is designed to be both easy to use and easy to extend.

[2] `https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/`

2. We compare several different algorithms on a wide range of graphs. The results of these comparisons were used to create a competitor for the first PACE challenge, where it scored third place in the exact sequential track and third place in the heuristic parallel track.
3. We show that two quite different practical algorithmic approaches – SAT solvers and FPT algorithms – work very well together for a wide range of instances. We thus propose to stimulate the exchange of techniques between those fields.

In Section 2, we give two equivalent formulations of tree width that we will use within this work. In the next section – Section 3 – we look at different algorithmic paradigms (constraint satisfaction problems, exact exponential algorithms, heuristics, and FPT algorithms) and compare several algorithms within those paradigms. We compare those algorithms on a wide range of different graphs and combine the best of them. The combined algorithm was submitted to the first PACE challenge, where it took the third place. Experimental comparison between the winners of the PACE challenge on almost 2.000 graphs are presented in Section 4. The results show that the developed approach is competitive. The appendix contains a compact overview on the graph sets used in our experimental comparisons.

1.2 Experimental Comparisons

Our experimental comparisons were designed in such a way that a *global overview* on the behaviour of the different algorithms is given. Our plots thus show *trends* that we have observed in our computational experiments. For example, a single algorithm may always beat another algorithm, two algorithms are largely incomparable, or an algorithm either terminates within a few seconds or never. While Jdrasil contains implementation of all of the discussed algorithms, this global overviews allowed us to decide upon a standard behaviour of the library. More detailed experiments on specific algorithms and their concrete evaluations can be found in the referenced papers.

2 Preliminaries

In this paper all graphs are undirected, simple, and connected unless stated otherwise. A *tree decomposition* of a graph G is a pair (T, ι) consisting of a tree T and a mapping ι from the nodes of T to subsets of vertices of G (called *bags*), such that (1) for every edge $\{u, v\} \in E(G)$ there is a node $x \in V(T)$ with $\{u, v\} \subseteq \iota(x)$, and (2) for all nodes $x, y, z \in V(T)$ we have $\iota(y) \subseteq \iota(x) \cap \iota(z)$ whenever y lies on the unique path between x and z in T.

The *width* of a tree decomposition is the maximum size of its bags minus 1, i. e., $\mathrm{width}(T, \iota) = \max_x\{\, |\iota(x)| - 1 \,\}$. The *tree width* of a graph G is the smallest width of any tree decomposition of G and is denoted as $\mathrm{tw}(G)$. Deciding whether a given graph G has tree width at most k is an NP-complete problem [1].

Note that the above definition of tree width does not immediately give rise to an algorithm that computes the optimal tree decomposition. It is, hence, useful to look at alternative characterizations of this concept:

An *elimination ordering* π of a graph $G = (V, E)$ is a bijection $\pi\colon V \to \{1, 2, \dots, |V|\}$. The *filled graph* $G_\pi = (V, E_\pi)$ of the elimination ordering π is a directed graph with edges E_π that are constructed via the following process:

- The first edge set E_π^0 simply equals E, where the edges are directed from the "lower" vertex (according to π) to the "higher" vertex, i. e.,

$$E_\pi^0 = \{\, (u, v) \mid \pi(u) < \pi(v) \wedge \{u, v\} \in E \,\}.$$

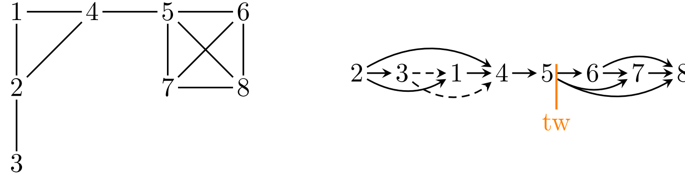


Figure 1 An example of a graph G and the corresponding filled graph G_π for the elimination ordering $\pi = 2, 3, 1, 4, 5, 6, 7, 8$. Here, the solid edges represent the edges of the original graph, while the dashed edges are the edges created by eliminating vertex 2.

- The next edge set E_π^{i+1} is generated by connecting all vertices u, v with $\pi(u) > i$ and $\pi(v) > i$ if both u and v are connected with the vertex $\pi^{-1}(i)$, i. e.,

$$E_\pi^i = E_\pi^{i-1} \cup \{\, (u,v) \mid \pi(v) > \pi(u) > i \wedge (\pi^{-1}(i), u) \in E_\pi^{i-1} \wedge (\pi^{-1}(i), v) \in E_\pi^{i-1} \,\}.$$

Finally, E_π is equal to $E_\pi^{|V|}$. Figure 1 shows an example of a graph G and the corresponding filled graph G_π.

The *width* of an elimination ordering π is the largest number of direct successors of a vertex in G_π, i. e., $\mathrm{width}(\pi) = \max_i\{\, |\{(u_i, v) \in E_\pi\}| \,\}$. The (optimal) width of the example on the right hand side is 3, as there exist three outgoing arcs from vertex 5.

The following fact is well known and allows us to characterize the tree width of a graph either via a suitable tree decomposition or via an elimination ordering.

▶ **Fact 1** (e. g. [14]). $\mathrm{tw}(G) = \max_\pi\{\mathrm{width}(\pi)\}$.

3 Computing Tree Decompositions

3.1 Point of View: Constraint Satisfaction Problem

A very common (theoretical and practical) approach to solve intractable problems is to first represent them as *constraint satisfaction problems* (CSP) and then solve those problems via specialized solvers. The most widely used solvers are SAT solvers that work on Boolean formulas or ILP solvers that work on linear inequalities. As it is typically not clear from the problem alone, which of those approaches leads to a better result, the next subsection compares experimental evaluations of both approaches on a certain formulation for the elimination ordering problem. The formulation we will use is based on the work of Berg and Järvisalo [7], which in turn is an improved version of a formulation of Samer and Veith [30].

3.1.1 The CSP formulation

If $G = (V, E)$ is a graph on $|V| = n$ vertices, our CSP first contains $n(n-1)/2$ variables $\mathrm{ord}_{i,j}$ for each $i \in \{1, \dots, n\}$ and each $j > i$, indicating that the vertex v_i appears before v_j in the elimination ordering. To simplify notation, for two integers i and j, let $\mathrm{ord}^*_{i,j}$ be either $\mathrm{ord}_{i,j}$ if $i < j$ or $\neg\mathrm{ord}_{j,i}$ if $j < i$. To ensure that these variables encode a linear ordering of the vertices, it is sufficient to enforce the transitivity: For all distinct $i, j, k \in \{1, \dots, n\}$, we need to ensure that if $\mathrm{ord}^*_{i,j}$ and $\mathrm{ord}^*_{j,k}$ are true, $\mathrm{ord}^*_{i,k}$ is also true.

To encode the directed edges of the filled graph G_π, another n^2 variables $\mathrm{arc}_{i,j}$ are introduced. As all original edges of G are present in G_π, for each $\{v_i, v_j\} \in E$, either $\mathrm{arc}_{i,j}$ or $\mathrm{arc}_{j,i}$ must be set. To be consistent with the ordering implied by $\mathrm{ord}_{i,j}$, we need to enforce that $\mathrm{ord}^*_{i,j}$ implies that $\mathrm{arc}_{j,i}$ is not set.

To describe the elimination process, note that if v_i and v_k are adjacent and v_i and v_j are adjacent with $\pi(i) < \pi(j)$ and $\pi(i) < \pi(k)$, the filled graph G_π contains either the arc (v_j, v_k) or the arc (v_k, v_j). Hence, if $\text{arc}_{i,j}$ and $\text{arc}_{i,k}$ are set and $\text{ord}^*_{j,k}$ is also set, we need to set $\text{arc}_{j,k}$ as well. To ensure that the width of the produced elimination does not exceed a value $t \in \mathbb{N}$, we also need to make sure that for each v_i, at most t edges (v_i, v_j) exist in G_π.

$\forall i,j,k \in \{1,\dots,n\}$	SAT formulation	ILP formulation
$i \neq j, i \neq k, j \neq k$	$\text{ord}^*_{i,j} \wedge \text{ord}^*_{j,k} \implies \text{ord}^*_{i,k}$	$\text{ord}^*_{i,k} - \text{ord}^*_{i,j} - \text{ord}^*_{j,k} \geq -1$
$\{v_i, v_j\} \in E$	$\text{arc}_{i,j} \vee \text{arc}_{j,i}$	$\text{arc}_{i,j} + \text{arc}_{j,i} \geq 1$
$i \neq j$	$\text{ord}^*_{i,j} \implies \neg\text{arc}_{j,i}$	$\text{ord}^*_{i,j} + \text{arc}_{j,i} \leq 2$
$i \neq j, i \neq k, j \neq k$	$\text{ord}^*_{j,k} \wedge \text{arc}_{i,j} \wedge \text{arc}_{i,k} \implies \text{arc}_{j,k}$	$\text{ord}^*_{j,k} + \text{arc}_{i,j} + \text{arc}_{i,k} - \text{arc}_{j,k} \leq 2$
	$\sum_{j=1}^{n} \text{arc}_{i,j} \leq t$	$\sum_{j=1}^{n} \text{arc}_{i,j} \leq t$
		$\text{arc}_{i,j}, \text{ord}_{i,j} \in \{0,1\}$

We extend this formulation by a trick observed in [12]: If $C \subseteq V$ is a clique in $G = (V, E)$, then there is an optimal elimination order π that eliminates C at last. Therefore, if we know a clique C, we can fix it at the end of the permutation. Of course, finding a clique of large cardinality is a difficult problem as well. We find them either with a CSP formulation as well, or, if this it not feasible, with a greedy heuristic. We noticed, however, that CSP solver performed very well on finding maximal cliques in graphs of small tree with – this is not surprising, as the cardinality of the largest clique is bounded by tree width of the graph. Given the clique C, we can extend the formulation as shown in the following table.

$\forall i,j \in \{1,\dots,n\}$	SAT	ILP
$i \in V \setminus C, j \in C$	$\text{ord}^*_{i,j}$	$\text{ord}^*_{i,j}$
$v_i \in C, v_j \in C$	$\text{ord}^*_{i,j}$	$\text{ord}^*_{i,j}$

The SAT formulation $\varphi(t)$ encodes a fixed value $t \in \mathbb{N}$. In order to determine $\text{tw}(G)$, the above encoding would be used for $t = n, n-1, \dots$ until the system does not have any solution, while the ILP would be able to minimize this quantity directly. We make use of the *iterative* abilities of modern SAT solvers that allows to add clauses to an already solved formula and thus does not require resets between the calls – this technique was also recommended by Berg and Järvisalo [7]. The SAT solver is thus able to reuse some of its already computed knowledge. We can thus solve $\varphi(n)$, add the constraints $\sum_{j=1}^{n} \text{arc}_{i,j} \leq n-1$ for each i, solve this new formula (which is equivalent to $\varphi(n-1)$, and repeat this process until $\varphi(t)$ is not satisfiable. Note that the last formula of the SAT formulation is a so called *cardinality constraint* and can be expressed, e. g., via sequential counters or sorting networks. See [4] for a discussion about this topic.

3.1.2 Experimental Evaluations

In order to determine whether a SAT solver or an ILP solver is more suited for finding the solution of our CSP, we performed a number of experimental evaluations. To solve the SAT formula, we made use of the SAT solver lingeling by Biere [8]. To solve the ILP, we used CPLEX of IBM [26]. Our test set was the set of 50 *easy instances* provided by the PACE challenge [28, 21]. The experimental results with a timeout of 5 minutes for each graph were very clear: While lingeling was able to solve 47 of the 50 instances within 14 seconds, CPLEX only managed to solve 23 of the graphs within 5 minutes. Furthermore, lingeling was faster on all of the provided graphs. A graphical representation of the experimental results can be found in Figure 2, where the graphs are sorted by the increasing running time of CPLEX. Here, the running time is shown on the y-axis in seconds.

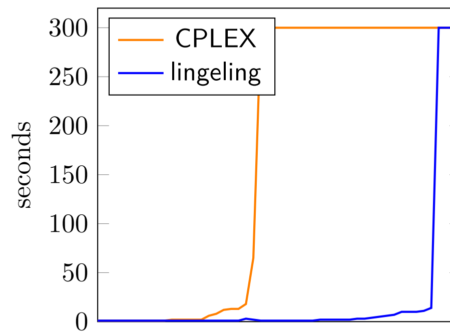


Figure 2 Comparing CPLEX and lingeling on the 50 easy instances of the PACE challenge with a timeout of 5 minutes.

3.2 Point of View: Exact Exponential Algorithms

One of the first algorithms to compute tree decompositions was presented by Arnborg et al. and is based on a $n^{O(\mathrm{tw}(G))}$ brute-force search [1]. Note that this algorithm is polynomial for constant tree width, but uses $O^*(2^n)$ time[3] and memory for non-constant tree width. A similar result can be achieved by combing a result of Seymour and Thomas [31], who have showed a connection of the tree width of the graph and a cops-and-robber search game, together with an algorithm by Berarducci and Intrigila [6], who provided an $n^{O(\mathrm{tw}(G))}$ algorithm to evaluate such games.

The characterization of tree width over elimination orders, as shown in the preliminaries, actually provides a much more rough brute-force approach: simply check all $n!$ possible permutations. It turns out that this strategy, combined with dynamic programming and some heuristics, can lead to $O^*(2^n)$ branch-and-bound algorithms. An example is QuickBB [24].

Finally, Bodlaender et al. have introduced a collection of Held-Karp like dynamic programs to compute optimal tree decompositions [12]. The practical feasible algorithms of this kind have time and space complexity $O^*(2^n)$ as well.

In the design of Jdrasil, it was interesting to study exact exponential time algorithms for two reasons: (1) to evaluate how competitive the SAT approach is against more direct approaches, and (2) to improve the SAT approach on certain instances. As usual for NP-hard problems, we can design instances on which certain algorithms do fail horribly. In our case, we noticed that the SAT approach fails on very symmetric instances. For example, it was not feasible to solve the McGee graph, which has only 24 vertices. We have implemented three different exact algorithms: A version of the cops-and-robber game, a QuickBB inspired branch-and-bound algorithm, and the dynamic program of [12]. Figure 3 shows the running time of one of the algorithms (from left to right: cops-and-robber (CAR), branch-and-bound (BAB), dynamic program (DP)) in blue, against the running time of a SAT solver (its time is shown in orange). The graphs are sorted by the running time of the SAT solver, therefore we see a phase transition in the orange plot (from feasible to unsolved within the time limit of 100 seconds). As we wish to improve the SAT approach, we are interested in the blue plot after the phase transition, i. e., in instances where the SAT solver fails. One can see in the very right plot that the dynamic program does not solve any of these instances and is thus not really useful for us. On the other hand, the cops-and-robber game (plot on the very left) and the branch-and-bound algorithm (center plot) do solve some instances on which the SAT

[3] The O^* notation does not only suppress constants, but also polynomial factors.

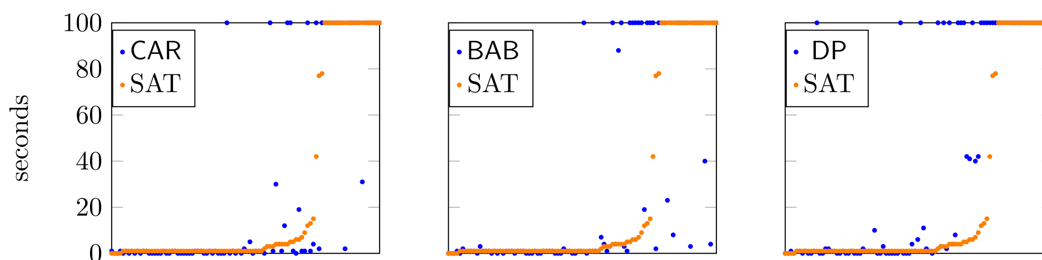


Figure 3 Comparison of cops-and-robber, branch-and-bound, and the dynamic program against the SAT solver on the 193 medium instances of the PACE challenge with a timeout of 100 seconds. The graphs are sorted by the running time of the SAT solver.

solver fails (whenever there is a blue peek after the phase transition). But on the down site, there are also a lot of instances where these algorithm fail, but SAT succeeds (blue peeks before the phase transition).

We conclude the following from the above experiment: First of all, the SAT approach is competitive, as it solves a couple of instances which are unsolved by all other algorithms. Second, there are instances that can be solved much quicker by the cops-and-robber game or the branch-and-bound algorithm and we face, therefore, the new problem of deciding when to use which algorithm.

3.3 Point of View: Upper and Lower Bounds

Research on heuristics for tree width is long standing and multi-faceted, including many different upper and lower bound algorithms. For an overview, we refer to the detailed survey papers by Bodleander and Koster [14, 15].

3.3.1 Upper bounds

The characterization of tree width over elimination orderings gives access to very simple, but still powerful, heuristics running in low-order polynomial running time (ranging from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^4)$ depending on the concrete heuristic). Recall that for graph $G = (V, E)$, every permutation π_V of V corresponds to a tree decomposition of G, and there is always one corresponding to an optimal decomposition. Many heuristics try to find such a permutation greedily: let π_S be a permutation of $S \subseteq V$, i. e., a partial permutation of V; the greedy algorithm selects a vertex $v \in V \setminus S$ that minimize some value function $\gamma(v)$ in the current graph $H = (V, E_{\pi_S})$ and appends v to the partial permutation π_S obtaining an updated permutation $\pi_{S\cup\{v\}}$. While many value functions γ are possible, an overview of six reasonable ones is given in [14]. They are summarized in Table 1, where $\psi_H(v) = |\{\, \{u,w\} \mid \{v,u\} \in E(H), \{v,w\} \in E(H), \{u,w\} \notin E(H) \,\}|$ equals the number of so called *fillin edges* and $\delta_H(v)$ is the degree of v in H.

We have implemented all of them to compare their quality. In the following plots, we sorted the graphs by their tree width (shown in orange), and have plotted the upper bounds produced by the heuristics in black. On the left picture of Figure 4, the heuristics with the six value functions from [14] are shown. We have omitted a labeling, because the message of this plot is not that a certain heuristic a is better on some instance x, but rather that they are all solid and that there is a lot of noise about which heuristic is better on which instance.

We used the result of the experiment to derive the heuristic that we now actually use. Note that, if we greedily select a vertex v that minimizes $\gamma(v)$, we may end up in a situation

Table 1 Value functions used by the upper bounds.

Name	$\gamma(v)$
Degree	$\delta_H(v)$
FillIn	$\psi_H(v)$
Degree+FillIn	$\delta_H(v) + \psi_H(v)$
SparsestSubgraph	$\psi_H(v) - \delta_H(v)$
FillInDegree	$\delta_H(v) + \psi_H(v)/n^2$
DegreeFillIn	$\delta_H(v) + \psi_H(v)/n$

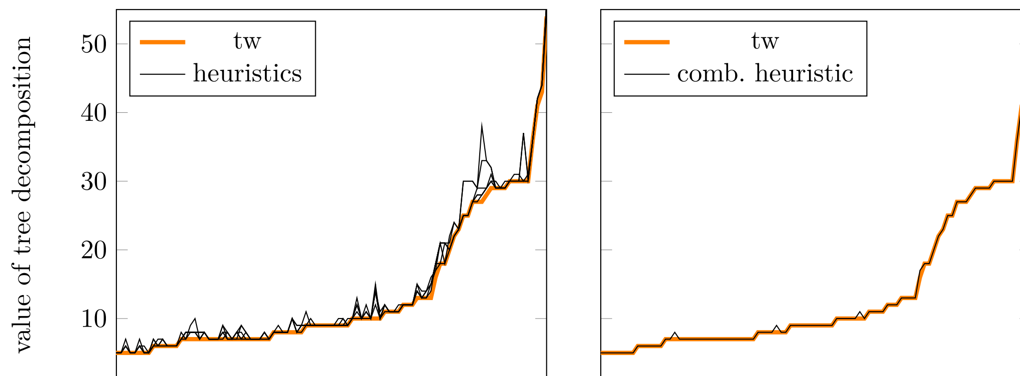


Figure 4 Comparison of the six upper bound heuristics against the optimal tree width on the 193 medium instances of the PACE challenge. The graphs are sorted by their tree width.

where we have a tie of more then one vertex. By breaking these ties randomly, we obtain a *randomized* algorithm. Already on very small test sets one can observe that the quality of this algorithm improves if we run it multiple times. On the other hand, if we repeat the algorithm multiple times, we do not have to fix a function γ. We have obtained very good results by running the heuristic $O(\sqrt{n})$ times and by selecting the value function γ in each run at random (from the pool of the six functions). An further improvement we did is a *look-ahead*: instead of choosing the vertex that minimizes γ, we take the vertex that minimizes the sum over the next c choices (for a constant c). With this extension (already for $c = 2$) we obtain the right plot from above. Note that of the 193 graphs of the test set, there are only 3 graphs on which the heuristic did not find the optimum.

3.3.2 Lower Bounds

There are a couple of very different approaches to compute lower bounds for the tree width of a graph. We refer to the second paper of Bodleander and Koster for an overview [15]. A promising approach is based on the fact that the tree width of every minor of a graph is bounded by the tree width of the host graph. Gogate and Dechter have developed a lower bound algorithm that greedily tries to find a minor with high tree width [24]. It repeatedly chooses a vertices v of minimum degree and one of its neighbors w and contracts the edge $\{v, w\}$. The largest minimum degree encountered in this process then yields a lower bound on the tree width. This algorithm can be used with different strategies concerning the greedy selection of the neighbour w that minimizes the value function $\gamma_v(w)$ in the current contracted graph H [15]. They are summarized in Table 2, where $N_H(v)$ denotes the neighbourhood of v in H.

Table 2 Value functions used by the lower bounds.

Name	$\gamma_v(w)$
min-d	$\delta_H(w)$
max-d	$-\delta_H(w)$
least-c	$\lvert N_H(v) \cap N_H(w) \rvert$

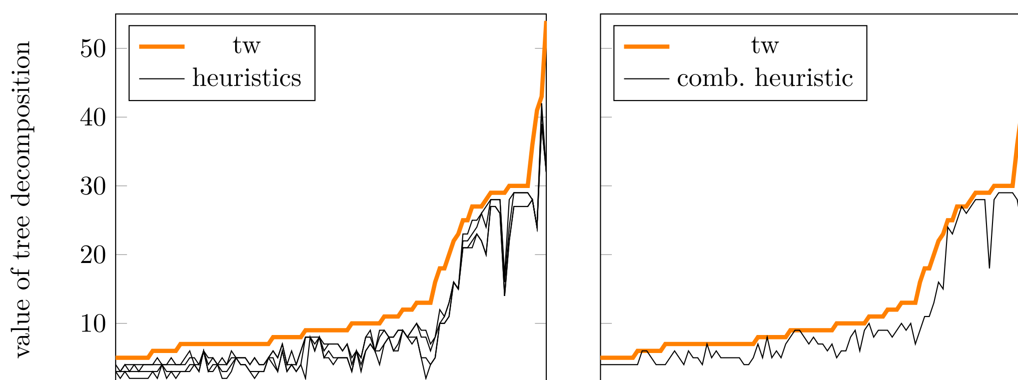


Figure 5 Comparison of the three lower bound heuristics against the optimal tree width on the 193 medium instances of the PACE challenge. The graphs are sorted by their tree width.

We have implemented the algorithm with the three strategies discussed in [15]. The results are shown in the following plots in Figure 5. The graphs are sorted by their tree width, which is plotted in orange. In the left plot, the lower bounds produced by the heuristic with the three different strategies is shown. We again omit the labels, as we wish to show the trend. One can see that the lower bounds have less quality then the upper bounds and that one strategy – least-c – actually dominates the others.

In [15] it is surveyed how the performance of a lower bound algorithm $\mathbb{A}$ can be boosted. The key idea is to work in the *k-neighbor improved graph*, which is obtained from the input graph by adding edges between all vertices that share k common neighbors. Starting with $k = \text{low}$, where low is the lower bound produced by $\mathbb{A}$ on the input graph G, we obtain a new graph G'. Then we can run $\mathbb{A}$ on this graph and eventually increase low allowing us to compute a new neighbor improved graph. Combined with the contraction idea of the algorithm of Gogate and Dechter, this yield a powerful lower bound algorithm (in [15] it is called LBN+). The quality of the produced lower bounds can be seen in the right plot above.

3.4 Point of View: Parameterized Complexity

The concept of tree width plays a central role in the field of parameterized complexity theory and has thus obtained a lot of attention in this area. While we typically model problems as languages $L \subseteq \Sigma^*$ over some fixed alphabet Σ, we define a *parameterized problem* as a tuple (Q, κ) with $Q \subseteq \Sigma^*$ and $\kappa\colon \Sigma^* \to \mathbb{N}$. The intuition is that the language Q models, as before, the problem, while κ (the parameter) highlights some special property of the instance. One can now analyze the running time of an algorithm for (Q, κ) with respect to both, the instance size and its parameter. We say a parameterized problem (Q, κ) is *fixed-parameter tractable* if there is an algorithm that decides for every $w \in \Sigma^*$ whether or not $w \in Q$ holds in time $f(\kappa(w)) \cdot \text{poly}(|w|)$, where $f\colon \mathbb{N} \to \mathbb{N}$ is some computable function. Note that we

can parameterize a problem in many different ways (number of vertices, maximum degree, solution size, girth, ...), and that some of the resulting parameterized problems may be fixed-parameter tractable while others may not.

Many NP- or even PSPACE-hard graph problems are fixed-parameter tractable with respect to the parameter tree width. A prime example is Courcell's Theorem [19], which states that all problems definable in monadic second-order logic can be solved in linear time, if a small tree decomposition is presented together with the input. This leads to the requirement of an algorithm that, given a graph $G = (V, E)$ with small tree width, computes an optimal tree decomposition of G. Such an algorithm was found by Bodlaender [9] and runs in time $f(\mathrm{tw}(G)) \cdot n$. Although this algorithms solves the problem theoretically, it can not be used in practice due to its huge constants [14]. Therefore, the parameterized complexity community has continued its search for a fast algorithm.

An important concept in parameterized complexity is *preprocessing*. Given an instance w of a parameterized problem (Q, κ), we wish to reduce it in polynomial time to a new instance w' with $|w'| \leq h(\kappa(w))$ for a computable function h. That is, we wish to reduce the problems to its hard core (the *kernel*), whose size may only depend on the parameter. This process is called *kernelization*. A positive result is that every fixed-parameter tractable problem has such a kernelization [18], and so does tree width. On the other hand, there are problems which probably do not have a kernel of polynomial size. Unfortunately, tree width is one of these problems [10]. The seek for good kernelization algorithms for tree width has lead to very efficient *heuristic reduction rules*, which safely reduce the graph but do not give any guarantees on their effectiveness [16]. A refined analysis of these reduction rules can be used to find polynomial kernels for tree width with respect to other parameters such as the vertex cover number or the size of a feedback vertex set [13].

Beside the effort of introducing good preprocessing algorithms, the parameterized complexity community has complemented the theoretically fast algorithm by Bodlaender [9] with actual fast *constant size approximation algorithms*. The first $4k + 3$ approximation algorithm running in time $O(3^{3k} \cdot n^2)$ was introduced by Robertson and Seymour during their quest to prove the graph minor theorem [29]. This was constantly improved by various authors (see [11] for an survey). The latest result is a $5k + 4$ algorithm running in time $2^{O(k)} \cdot n$.

For an implementation that should compute an optimal tree decomposition, an approximation algorithm as the one by Robertson and Seymour can be interesting in two ways: it produces lower *and* upper bounds at the same time. We have implemented an algorithm that is inspired by the one of Robertson and Seymour (see the textbook [20] for details), and have analyzed its practical running time and the quality of the produced lower and upper bounds. The following two plots in Figure 6 show the results of our experiments. On the left, we have plotted the running time (in seconds) of the approximation algorithm in blue against the running time of the SAT approach in orange. Here, the graphs are again sorted by the running time of the SAT solver and, hence, we have a phase transition in the orange plot from feasible to not feasible. The algorithm performs quite well and, in particular, does only fail on very few instances that can be handled by the SAT solver. The plot on the right shows the computed lower and upper bounds (in blue) against the exact tree width of the graphs (in orange). Here, the graphs are sorted by their tree width. While these bounds are reasonable – as expected from an approximation algorithm – they come short in comparison to the lower and upper bounds described in the last section.

From the experiments we conclude that the approximation algorithm delivers, besides its theoretically beauty, a practical access to the computation of tree decompositions. However, it falls short against the other algorithms in our tool box and is thus not used in our final version.

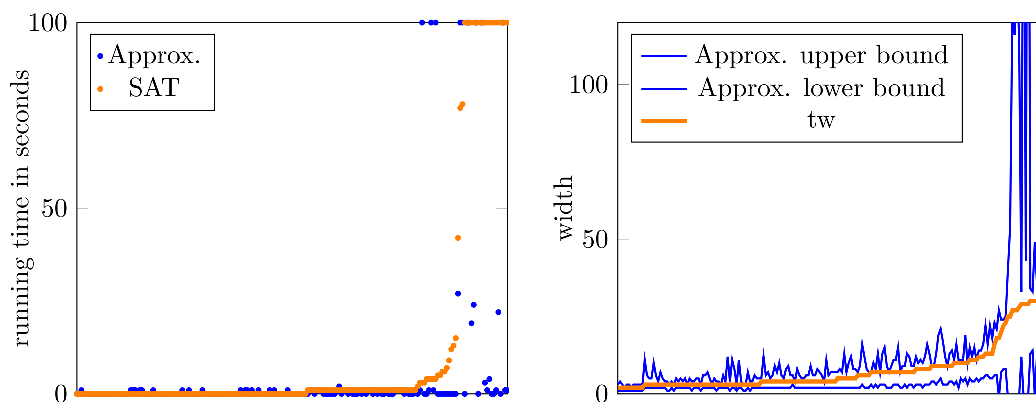


Figure 6 Comparison of the approximation algorithm and the SAT solver on the 193 medium instances of the PACE challenge with a timeout of 100 seconds. On the left, the graphs are sorted by the running time of the SAT solver and by their tree width on the right.

3.5 Cherry-Pick the Best from each World

While Jdrasil is designed as library and provides access to all implemented algorithms mentioned above, we have do decide which of these we actually wish to use if we want to compute an exact tree decomposition. The core of our algorithm is the SAT approach and we try to use it whenever possible. It is known that different SAT solvers behave differently on specific formulas and we, thus, tested different solvers on our formulas. It turned out that the SAT solver glucose of Audemard and Simon [3] – based on the classical solver MiniSat by Eén and Sörensson [23] – outperforms lingeling in our setting. We thus chose this solver for use in our final version.

We use the preprocessing techniques developed by the FPT community as reducing the instance almost always makes sense. This is in particular important from the view point of using a SAT solver, as the SAT solver does handle a huge tree in the same way it handles every other graph. The parameterized point of view also influenced the design of the formula, which uses many cardinality constraints. These constraints are usually implemented at the cost of $O(n \log n)$ auxiliary variables. However, we can also implement them using $O(kn)$ auxiliary variables, where k is the tree width of the graph, i. e., we craft the formula with respect to a parameter.

We use the lower and upper bound heuristics presented to prune the search space and to reduce the number of formulas that we have to consider. From the different exact exponential time algorithms, we choose to use the cops-and-robber game and the SAT approach. While the cops-and-robber game *always* runs in time $n^{\mathcal{O}(\mathrm{tw}(G))}$, the behaviour of the SAT solver is much more diverse. While it is able to solve `contiki_lpp_send_probe.gr` – a graph containing 92 vertices – within 54 seconds, it is not able solve the McGee graph containing 24 vertices. In contrast to this, the cops-and-robber game solve the McGee graph within a second. Our computational experiments showed that the cops-and-robber outperformed the other approaches for those graphs with treewidth at most 8 or at most 25 vertices. We thus use the cops-and-robber game, if our computed upper bound is at most 8 or if the graph has at most 25 vertices. On the other graphs, we use the SAT solver. Note that, of course, this decision is made after preprocessing, i. e., depending on the size of the "hard core" of the graph. The performance of the complete algorithm is analyzed in the next section.

4 Experimental Results

4.1 PACE 2016

As noted in the introduction, the Parameterized Algorithms and Computational Experiments (PACE) challenge was started in 2016 to "investigate the applicability of algorithmic ideas studied and developed in the subfields of multivariate, fine-grained, parameterized, or fixed-parameter tractable algorithms" [28, 21]. The challenge consisted of two tracks: one for tree width and one for feedback vertex set. We submitted the algorithm described in Section 3.5 to the the exact sequential competition of the tree width track (i. e. algorithms needed to output the optimal tree decomposition without the use of parallelism). The programs were given 200 graphs with predetermined timeouts ranging between 100 seconds an 3600 seconds. Our program (Jdrasil [5]) took the third place and solved 166 of the graphs. The second place was awarded to the program BZTreewidth [17] of Hans Bodlaender and Tom van der Zanden that solved 173 instances. They used a combination of dynamic programming and balanced separators. Finally, the first place was awarded to the program Exact treewidth [33] by Hisao Tamaki that solved 199 instances and used an improved version of the algorithm of Arnborg et al. [1].

4.2 Graph Benchmarks

The authors of this work later found a subtle bug in script configuring their implementation of Jdrasil that scheduled all simultaneously running instance of Jdrasil on the same processor. To test the feasibility of our approach, we ran the three winners of the PACE challenge on a benchmark set of 1813 graphs with a timeout of 300 seconds on each graph. Note that these experiments were performed in order to test the feasibility of our cherry-picking approach on a wide range of graphs within a reasonable time frame. They are *not* intended to be a replacement of the experiments determining the winner of the PACE challenge (where sophisticated voting rules were used to determine the winner). This yielded the following results:

	number of solved instances	average running time
Jdrasil [5]	1188	5,48 seconds
BZTreewidth [17]	1004	7,83 seconds
Exact treewidth [33]	1307	4,53 seconds

In order to understand the different behaviours of the algorithms, we have also looked at all combinations (p_1, p_2) of programs and the instances they were not able to solve. In the following table, an entry $x/y/z$ in the row labeled with p_1 and in the column labeled with p_2 denotes that the number of instances that p_1 and p_2 did both not solve was x, the number of instances that p_1 did not solve, but p_2 did, was y, and the number of instances that p_2 did not solve, but p_1 did, was z. For example, the colored entry shows that there were 505 instances, which neither Jdrasil nor BZTreewidth solved, only 30 instances that Jdrasil did not solve, but BZTreewidth solved, and 188 instances that Jdrasil solved, but BZTreewidth did not.

	Jdrasil	BZTreewidth	Exact treewidth
Jdrasil	–	505/30/188	329/206/96
BZTreewidth	505/188/30	–	397/296/28
Exact treewidth	329/96/209	397/28/296	–

Besided the average running time of the programs, another important aspect is the number of graphs where p_1 was faster than p_2. In the following table, an entry $x/y/z$ in the row labeled with p_1 and in the column labeled with p_2 denotes that p_1 was faster than p_2 on x graphs, while p_1 was faster than p_2 on y graphs, and z denotes the seconds p_1 was faster, minus the seconds p_2 was faster, i. e., if it is positive, p_1 was faster in total. This is an important information: It could be the case that p_1 is a second faster on half of the instances, but that p_2 is multiple minutes faster on a quarter of the remaining instances. While p_1 would outperform p_2 concerning the number of instances it solved faster, p_2 is clearly preferable. For example, the colored entry shows that Jdrasil was faster than BZTreewidth on 280 instances, while BZTreewidth outperformed Jdrasil on 236 instances. In total, Jdrasil outperformed BZTreewidth by 10231 seconds.

	Jdrasil	BZTreewidth	Exact treewidth
Jdrasil	–	280/236/10231	16/590/ − 6077
BZTreewidth	236/280/ − 10231	–	15/387/ − 12214
Exact treewidth	590/16/6077	387/15/12214	–

In summary, we believe that the results of this section show that the “cherry-picking”-approach described in the last section is competitive even to very specialized implementations. Our program Jdrasil solved more instances than the second place winner BZTreewidth and is substantially faster in total.

5 Handle the Use Cases

Case Example A: In order to compute the tree width of the causal graph, the researcher downloads Jdrasil from its homepage [5] and enters the command `$ ./gradlew exact` to build the scripts `tw-exact` (for Unix) or `tw-exact.bat` (for Windows). If her graph is stored in the file `graph.gr` (either in the PACE format [28, 21] or in the DIMACS graph format [22]), she can compute an optimal tree decomposition with the following command: `$ ./tw-exact < graph.gr`

Case Example B: The professor tells the student to look into the Jdrasil manual, which can be generated by the command `$ ./gradlew manual` that produces the manual in the directory `build/docs/manual`. After careful reading, he writes the following Java code that uses Jdrasil to compute an exact tree decomposition:

```
import jdrasil.algorithms.ExactDecomposer;
import jdrasil.graph.Graph;
import jdrasil.graph.TreeDecomposition;

public class Algorithm {
    public int computeFirstAlgorithm(Graph<Integer> g) {
        TreeDecomposition<Integer> decomposition = null;
        try {
          ExactDecomposer<Integer> ex = new ExactDecomposer<>(g);
          decomposition = ex.call();
        }
        ...
```

He can compile and run this code by including the file `Jdrasil.jar` produced by `$ ./gradlew jar` which can be found in the directory `build/jars`.

Case Example C: The Ph. D. student looks at the documentation of the java code generated by `$ ./gradlew javadoc` and finds the generated HTML files in the directory `build/docs/javadoc`. After considering which classes she needs, she decides to make use of the class `graph.Graph` (to use an efficient implementation of the underlying graph), the class `algorithms.lowerbounds.MinorMinWidthLowerbound` (to compute a lower bound) and the class `algorithms.preprocessing.GraphReducer` (to reduce the graph by using the reduction rules of [16]).

6 Conclusion

In this paper we have presented our Java library Jdrasil for the computation of tree decompositions. The goals we have achieved with the library are threefold: first of all we hope that the library gives algorithmic engineers, who wish to work on tree decompositions, an easy access to these complex graph theoretic structures (Section 5). On the other hand, Jdrasil implements a broad range of tools that can be used by theorists or engineers that wish to implement new algorithms for computing tree decompositions (Section 3). Our computational results imply that different algorithms are needed for different graphs and we show that combining several of those algorithms allows us to be competitive against other optimized implementations (Section 4). This "cherry-picking" can be done easily due to the highly modular design of Jdrasil.

All together, we hope that Jdrasil will be helpful for studying tree decompositions both in a theoretical and practical domain; and we look towards to further improve and extend the implementation.

References

1 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987. doi:10.1137/0608024.

2 Stefan Arnborg and Andrzej Proskurowski. Linear Time Algorithms for NP-hard Problems Restricted to Partial k-Trees. *Discrete applied mathematics*, 23(1):11–24, 1989. doi:10.1016/0166-218X(89)90031-0.

3 Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proc. IJCAI*, pages 399–404, 2009.

4 Olivier Bailleux and Yacine Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In *Proc. CP*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.

5 Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil, 2016. URL: https://github.com/maxbannach/Jdrasil.

6 Alessandro Berarducci and Benedetto Intrigila. On the Cop Number of a Graph. *Advances in Applied Mathematics*, 14(4):389–403, 1993. doi:10.1006/aama.1993.1019.

7 Jeremias Berg and Matti Järvisalo. SAT-Based Approaches to Treewidth Computation: An Evaluation. In *Proc. ICTAI*, pages 328–335. IEEE Computer Society, 2014. doi:10.1109/ICTAI.2014.57.

8 Armin Biere. Lingeling, Plingeling, Picosat and Precosat at SAT Race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

9 Hans L. Bodlaender. A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth. In *Proc. STOC*, pages 226–234. ACM, 1993. doi:10.1145/167088.167161.

10 Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. On Problems Without Polynomial Kernels. *Journal of Computer and System Sciences*, 75(8):423–434, 2009. doi:10.1016/j.jcss.2009.04.001.

11 Hans L. Bodlaender, Pal Gronas Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. An $O(c^k n)$ 5-Approximation Algorithm for Treewidth. In *Proc. FOCS*, pages 499–508, Oct 2013. doi:10.1109/FOCS.2013.60.

12 Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On Exact Algorithms for Treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, 2012. doi:10.1145/2390176.2390188.

13 Hans L. Bodlaender, Bart M. P. Jansen, and Stefan Kratsch. Preprocessing for Treewidth: A Combinatorial Analysis through Kernelization. In *Proc. ICALP*, volume 6755 of *Lecture Notes in Computer Science*, pages 437–448. Springer, 2011. doi:10.1137/120903518.

14 Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth Computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010. doi:10.1016/j.ic.2009.03.008.

15 Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations II. Lower Bounds. *Information and Computation*, 209(7):1103–1119, 2011. doi:10.1016/j.ic.2011.04.003.

16 Hans L. Bodlaender, Arie M. C. A. Koster, and Frank van den Eijkhof. Pre-Processing for Triangulation of Probabilistic Networks. *Computational Intelligence*, 21(3):286–305, 2005. doi:10.1111/j.1467-8640.2005.00274.x.

17 Hans L. Bodlaender and Tom van der Zanden. BZTreewidth, 2016. URL: https://github.com/TomvdZanden/BZTreewidth.

18 Liming Cai, Jianer Chen, Rodney G. Downey, and Michael R. Fellows. Advice Classes of Parameterized Tractability. *Ann. Pure Appl. Logic*, 84(1):119–138, 1997. doi:10.1016/S0168-0072(95)00020-8.

19 Bruno Courcelle. Graph Rewriting: An Algebraic and Logic Approach. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 193–242. Elsevier, Amsterdam, Netherlands and MIT Press, Cambridge, Massachusetts, 1990. doi:10.1016/B978-0-444-88074-1.50010-X.

20 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2015.

21 Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *LIPIcs*, pages 30:1–30:9. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.IPEC.2016.30.

22 DIMACS Graph Format. Accessed: 2017-01-26. URL: http://prolland.free.fr/works/research/dsat/dimacs.html.

23 Niklas Eén and Niklas Sörensson. An Extensible SAT-Solver. In *Proc. SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

24 Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. In *Proc. UAI*, pages 201–208. AUAI Press, 2004.

25 Thore Husfeldt and Iyad A. Kanj, editors. *Proc. IPEC*, volume 43 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015.

26 IBM. *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual*. URL: https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.1/ilog.odms.studio.help/pdf/usrcplex.pdf.

27 David R. Karger and Nathan Srebro. Learning Markov Networks: Maximum Bounded Tree-Width Graphs. In *Proc. SODA*, pages 392–401. ACM/SIAM, 2001.

28 The Parameterized Algorithms and Computational Experiments Challenge (PACE). Accessed: 2017-01-26. URL: https://pacechallenge.wordpress.com/.
29 Neil Robertson and Paul D. Seymour. Graph Minors. XIII. The Disjoint Paths Problem. *Journal of Combinatorial Theory*, 63(1):65–110, 1995. doi:10.1007/978-3-540-24605-3_37.
30 Marko Samer and Helmut Veith. Encoding Treewidth into SAT. In *Proc. SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 45–50. Springer, 2009. doi:10.1007/978-3-642-02777-2_6.
31 Paul D. Seymour and Robin Thomas. Graph Searching and a Min-Max Theorem for Tree-Width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993. doi:10.1006/jctb.1993.1027.
32 Yinglei Song, Chunmei Liu, Russell L. Malmberg, Fangfang Pan, and Liming Cai. Tree Decomposition Based Fast Search of RNA Structures Including Pseudoknots in Genomes. In *Proc. CSB*, pages 223–234. IEEE Computer Society, 2005.
33 Hisao Tamaki. Exact treewidth, 2016. URL: https://github.com/TCS-Meiji/treewidth-exact.

A Graph Benchmarks

The 50 easy instances used in Section 3.1 are taken from the easy instances provided by the PACE challenge. The notation $(n/m/t)$ means that the graph has n vertices, m edges and tree width t.

- `BalancedTree_3,5.gr` (364/363/1)
- `contiki_collect_send_next_packet.gr` (26/25/1)
- `contiki_ctk_ctk_menu_add.gr` (25/27/2)
- `contiki_cxmac_input_packet.gr` (90/97/3)
- `contiki_dhcpc_dhcpc_init.gr` (34/34/2)
- `contiki_dhcpc_dhcpc_request.gr` (27/27/2)
- `contiki_httpd-cfs_send_file.gr` (44/48/3)
- `contiki_ifft_ifft.gr` (172/180/2)
- `contiki_ircc_list_channel.gr` (70/76/3)
- `contiki_lpp_send_packet.gr` (116/120/2)
- `contiki_nullrdc_packet_input.gr` (28/30/3)
- `contiki_polite-announcement_send_timer.gr` (31/31/2)
- `contiki_powertrace_add_stats.gr` (46/47/2)
- `contiki_powertrace_powertrace_print.gr` (323/323/2)
- `contiki_profile_profile_episode_start.gr` (31/32/2)
- `contiki_psock_psock_generator_send.gr` (61/68/4)
- `contiki_ringbuf_ringbuf_put.gr` (29/29/2)
- `contiki_rudolph0_send_nack.gr` (27/26/1)
- `contiki_rudolph1_rudolph1_send.gr` (30/29/1)
- `contiki_runicast_runicast_open.gr` (24/23/1)
- `contiki_shell-collect-view_process_thread_collect_view_data_process.gr` (61/62/2)
- `contiki_shell-rime-ping_recv_mesh.gr` (47/47/2)
- `contiki_shell-rime_recv_collect.gr` (62/64/2)
- `contiki_shell-text_process_thread_shell_echo_process.gr` (25/25/2)
- `contiki_shell_shell_register_command.gr` (42/45/2)
- `contiki_tcpip_eventhandler.gr` (98/112/2)
- `contiki_uip-neighbor_uip_neighbor_add.gr` (67/71/3)
- `contiki_uip-over-mesh_recv_data.gr` (85/88/2)
- `contiki_uip_uip_init.gr` (26/27/2)
- `contiki_webclient_senddata.gr` (108/109/2)

- fuzix_clock_settime_clock_settime.gr (20/21/2)
- fuzix_devf_fd_transfer.gr (119/129/3)
- fuzix_devio_bfind.gr (27/29/3)
- fuzix_devio_kprintf.gr (69/78/3)
- fuzix_difftime_difftime.gr (74/73/1)
- fuzix_fgets_fgets.gr (53/58/3)
- fuzix_filesys_filename.gr (45/48/3)
- fuzix_filesys_getinode.gr (52/57/3)
- fuzix_filesys_i_open.gr (129/143/3)
- fuzix_filesys_newfstab.gr (20/21/2)
- fuzix_getpass__gets.gr (31/35/3)
- fuzix_malloc___insert_chunk.gr (104/116/3)
- fuzix_process_getproc.gr (32/35/2)
- fuzix_ran_rand.gr (46/48/2)
- fuzix_regexp_regcomp.gr (118/129/2)
- fuzix_se_ycomp.gr (83/96/3)
- fuzix_stat_statfix.gr (52/51/1)
- fuzix_syscall_fs2_chown_op.gr (27/28/2)
- fuzix_tty_tty_read.gr (123/137/4)

The 193 instances used in Section 3.2, Section 3.3, and in Section 3.4 are taken from the 100 second instances of the PACE challenge. The notation (n/m) means that the graph has n vertices and m edges.

- AhrensSzekeresGeneralizedQuadrangleGraph_3.gr (27/135)
- BalancedTree_3,5.gr (364/363)
- BlanusaSecondSnarkGraph.gr (18/27)
- ChvatalGraph.gr (12/24)
- ClebschGraph.gr (16/40)
- CycleGraph_100.gr (100/100)
- DesarguesGraph.gr (20/30)
- DodecahedralGraph.gr (20/30)
- DorogovtsevGoltsevMendesGraph.gr (3282/6561)
- DoubleStarSnark.gr (30/45)
- DyckGraph.gr (32/48)
- ErreraGraph.gr (17/45)
- FibonacciTree_10.gr (143/142)
- FlowerSnark.gr (20/30)
- FolkmanGraph.gr (20/40)
- FriendshipGraph_10.gr (21/30)
- GNP_20_10_0.gr (20/28)
- GNP_20_10_1.gr (20/24)
- GNP_20_20_0.gr (20/46)
- GNP_20_20_1.gr (20/48)
- GNP_20_30_0.gr (20/56)
- GNP_20_30_1.gr (20/63)
- GNP_20_40_0.gr (20/78)
- GNP_20_40_1.gr (20/71)
- GNP_20_50_0.gr (20/91)
- GNP_20_50_1.gr (20/106)
- GeneralizedPetersenGraph_10_4.gr (20/30)
- GoethalsSeidelGraph_2_3.gr (16/72)
- GoldnerHararyGraph.gr (11/27)
- GrayGraph.gr (54/81)
- GrotzschGraph.gr (11/20)

- HararyGraph_6_15.gr (15/45)
- HeawoodGraph.gr (14/21)
- HoffmanGraph.gr (16/32)
- HyperStarGraph_10_2.gr (45/72)
- IcosahedralGraph.gr (12/30)
- KneserGraph_10_2.gr (45/630)
- LadderGraph_20.gr (40/58)
- MarkstroemGraph.gr (24/36)
- McGeeGraph.gr (24/36)
- MeredithGraph.gr (70/140)
- NauruGraph.gr (24/36)
- NonisotropicOrthogonalPolarGraph_3_5.gr (15/60)
- NonisotropicUnitaryPolarGraph_3_3.gr (63/1008)
- OddGraph_4.gr (35/70)
- OrthogonalArrayBlockGraph_4_3.gr (9/36)
- PaleyGraph_17.gr (17/68)
- PappusGraph.gr (18/27)
- PoussinGraph.gr (15/39)
- RKT_20_40_10_0.gr (20/87)
- RKT_20_40_10_1.gr (20/87)
- RKT_20_50_10_0.gr (20/73)
- RKT_20_50_10_1.gr (20/73)
- RKT_20_60_10_0.gr (20/58)
- RKT_20_60_10_1.gr (20/58)
- RKT_20_70_10_0.gr (20/44)
- RKT_20_70_10_1.gr (20/44)
- RKT_20_80_10_0.gr (20/29)
- RKT_20_80_10_1.gr (20/29)
- RandomBarabasiAlbert_100_2.gr (100/196)
- RandomBipartite_10_50_3.gr (60/138)
- RandomGNM_100_100.gr (100/100)
- RingedTree_6.gr (63/123)
- SchlaefliGraph.gr (27/216)
- ShrikhandeGraph.gr (16/48)
- SierpinskiGasketGraph_3.gr (15/27)
- SquaredSkewHadamardMatrixGraph_2.gr (49/588)
- StarGraph_100.gr (101/100)
- SylvesterGraph.gr (36/90)
- SzekeresSnarkGraph.gr (50/75)
- TaylorTwographDescendantSRG_3.gr (27/135)
- TaylorTwographSRG_3.gr (28/210)
- Toroidal6RegularGrid2dGraph_4_6.gr (24/72)
- WheelGraph_100.gr (100/198)
- WorldMap.gr (166/323)
- contiki_calc_input_to_operand1.gr (31/33)
- contiki_collect_enqueue_dummy_packet.gr (46/46)
- contiki_collect_received_announcement.gr (52/59)
- contiki_collect_send_ack.gr (53/52)
- contiki_collect_send_next_packet.gr (26/25)
- contiki_collect_send_queued_packet.gr (95/99)
- contiki_contikimac_input_packet.gr (116/127)
- contiki_contikimac_powercycle.gr (166/194)
- contiki_ctk_ctk_menu_add.gr (25/27)
- contiki_cxmac_input_packet.gr (90/97)
- contiki_dhcpc_dhcpc_init.gr (34/34)

- contiki_dhcpc_dhcpc_request.gr (27/27)
- contiki_dhcpc_handle_dhcp.gr (276/313)
- contiki_httpd-cfs_send_file.gr (44/48)
- contiki_httpd-cfs_send_headers.gr (106/116)
- contiki_ifft_ifft.gr (172/180)
- contiki_ircc_handle_connection.gr (138/161)
- contiki_ircc_list_channel.gr (70/76)
- contiki_lpp_dutycycle.gr (102/114)
- contiki_lpp_init.gr (22/21)
- contiki_lpp_send_packet.gr (116/120)
- contiki_lpp_send_probe.gr (92/94)
- contiki_nullrdc_packet_input.gr (28/30)
- contiki_polite-announcement_send_timer.gr (31/31)
- contiki_powertrace_add_stats.gr (46/47)
- contiki_powertrace_powertrace_print.gr (323/323)
- contiki_process_exit_process.gr (72/82)
- contiki_profile_profile_episode_start.gr (31/32)
- contiki_psock_psock_generator_send.gr (61/68)
- contiki_psock_psock_readto.gr (56/61)
- contiki_ringbuf_ringbuf_put.gr (29/29)
- contiki_route-discovery_route_discovery_discover.gr (20/20)
- contiki_rudolph1_rudolph1_open.gr (27/26)
- contiki_rudolph1_write_data.gr (35/36)
- contiki_serial-line_process_thread_serial_line_process.gr (72/81)
- contiki_shell-base64_base64_add_char.gr (70/74)
- contiki_shell-collect-view_process_thread_collect_view_data_process.gr (61/62)
- contiki_shell-netperf_memcpy_misaligned.gr (30/32)
- contiki_shell-ps_process_thread_shell_ps_process.gr (45/46)
- contiki_shell-rime-debug_recv_broadcast.gr (24/23)
- contiki_shell-rime-ping_recv_mesh.gr (47/47)
- contiki_shell-rime_process_thread_shell_send_process.gr (89/95)
- contiki_shell-rime_recv_collect.gr (62/64)
- contiki_shell-sendtest_read_chunk.gr (30/32)
- contiki_shell-text_process_thread_shell_echo_process.gr (25/25)
- contiki_shell_process_thread_shell_server_process.gr (76/85)
- contiki_shell_shell_register_command.gr (42/45)
- contiki_tcpip_eventhandler.gr (98/112)
- contiki_uip-neighbor_uip_neighbor_add.gr (67/71)
- contiki_uip-neighbor_uip_neighbor_periodic.gr (20/21)
- contiki_uip-over-mesh_recv_data.gr (85/88)
- contiki_uip_uip_connect.gr (111/120)
- contiki_uip_uip_init.gr (26/27)
- contiki_uip_uip_unlisten.gr (19/20)
- contiki_webclient_senddata.gr (108/109)
- contiki_webclient_webclient_appcall.gr (98/111)
- dimacs_anna.gr (138/260)
- dimacs_fpsol2.i.3.gr (206/2645)
- dimacs_inithx.i.2.gr (299/5162)
- dimacs_inithx.i.2-pp.gr (220/4165)
- dimacs_inithx.i.3-pp.gr (196/2185)
- dimacs_jean.gr (77/184)
- dimacs_miles1000.gr (128/1594)
- dimacs_miles250.gr (125/241)
- dimacs_miles750.gr (128/1252)
- dimacs_mulsol.i.1.gr (100/1725)

- dimacs_mulsol.i.2.gr (101/1233)
- dimacs_mulsol.i.3.gr (102/1233)
- dimacs_mulsol.i.4-pp.gr (78/1062)
- dimacs_mulsol.i.5.gr (102/1224)
- dimacs_mulsol.i.5-pp.gr (77/974)
- dimacs_myciel5.gr (46/139)
- dimacs_queen5_5.gr (25/106)
- dimacs_queen6_6.gr (36/217)
- dimacs_queen7_7.gr (49/388)
- dimacs_zeroin.i.2.gr (85/951)
- dimacs_zeroin.i.3.gr (83/917)
- dimacs_zeroin.i.3-pp.gr (49/651)
- fuzix_abort_abort.gr (21/20)
- fuzix_bankfixe_pagemap_alloc.gr (21/22)
- fuzix_clock_gettime_clock_gettime.gr (39/40)
- fuzix_clock_gettime_div10quickm.gr (30/29)
- fuzix_clock_settime_clock_settime.gr (20/21)
- fuzix_devf_fd_transfer.gr (119/129)
- fuzix_devio_bfind.gr (27/29)
- fuzix_devio_kprintf.gr (69/78)
- fuzix_difftime_difftime.gr (74/73)
- fuzix_fgets_fgets.gr (53/58)
- fuzix_filesys_filename.gr (45/48)
- fuzix_filesys_getinode.gr (52/57)
- fuzix_filesys_i_open.gr (129/143)
- fuzix_filesys_newfstab.gr (20/21)
- fuzix_filesys_srch_mt.gr (31/33)
- fuzix_gethostname_gethostname.gr (30/31)
- fuzix_getpass__gets.gr (31/35)
- fuzix_inode_rwsetup.gr (77/83)
- fuzix_malloc___insert_chunk.gr (104/116)
- fuzix_nanosleep_clock_nanosleep.gr (110/121)
- fuzix_process_getproc.gr (32/35)
- fuzix_qsort__lqsort.gr (89/94)
- fuzix_ran_rand.gr (46/48)
- fuzix_readdir_readdir.gr (60/65)
- fuzix_regexp_regcomp.gr (118/129)
- fuzix_se_ycomp.gr (83/96)
- fuzix_setbuffer_setbuffer.gr (43/44)
- fuzix_setenv_setenv.gr (122/131)
- fuzix_stat_statfix.gr (52/51)
- fuzix_syscall_fs2__fchdir.gr (22/22)
- fuzix_syscall_fs2_chown_op.gr (27/28)
- fuzix_syscall_proc__time.gr (48/49)
- fuzix_sysconf_sysconf.gr (142/162)
- fuzix_tty_tty_read.gr (123/137)
- fuzix_usermem_ugets.gr (24/25)
- fuzix_vfscanf_vfscanf.gr (587/668)
- stdlib_gmtime.gr (117/123)
- stdlib_mktime.gr (93/97)
- stdlib_print_format.gr (544/609)
- stdlib_sincoshf.gr (110/117)

The 1813 graphs used in Section 4 come from a wide range of sources: All of the graphs (exact and heuristic) from the PACE challenge, randomly generated partial k-trees, large grids, coloring instances[4], and classical test instances for tree width[5].

B Technical Specifications

All of the experiments were performed on a machine with 64 cores, where each core is a 2.1 Gigahertz processor. Note that we only used a single core for all experiments in order to prevent parallel programs from having an unfair advantage. The machine has 128 Gigabyte RAM and runs openSUSE 13.1 (Bottle) with kernel 3.11.10-29-desktop.

[4] Found at `http://mat.gsia.cmu.edu/COLOR/instances.html`.
[5] Found at `https://github.com/FrankvH/BooleanWidth/tree/master/Graphs/tw-lib`.

On the Separation of Topology-Free Rank Inequalities for the Max Stable Set Problem

Stefano Coniglio[1] **and Stefano Gualandi**[2]

1 **Department of Mathematical Sciences, University of Southampton, Southampton, UK**
s.coniglio@soton.ac.uk

2 **Department of Mathematics, University of Pavia, Pavia, Italy**
stefano.gualandi@unipv.it

Abstract

In the context of finding the largest stable set of a graph, rank inequalities prescribe that a stable set can contain, from any induced subgraph of the original graph, at most as many vertices as the stability number of the former. Although these inequalities subsume many of the valid inequalities known for the problem, their exact separation has only been investigated in few special cases obtained by restricting the induced subgraph to a specific topology.

In this work, we propose a different approach in which, rather than imposing topological restrictions on the induced subgraph, we assume the right-hand side of the inequality to be fixed to a given (but arbitrary) constant. We then study the arising separation problem, which corresponds to the problem of finding a maximum weight subgraph with a bounded stability number. After proving its hardness and giving some insights on its polyhedral structure, we propose an exact branch-and-cut method for its solution. Computational results show that the separation of topology-free rank inequalities with a fixed right-hand side yields a substantial improvement over the bound provided by the fractional clique polytope (which is obtained with rank inequalities where the induced subgraph is restricted to a clique), often better than that obtained with Lovász's Theta function via semidefinite programming.

1998 ACM Subject Classification G.1.6 [Optimization] Integer Programming

Keywords and phrases Maximum stable set problem, rank inequalities, cutting planes, integer programming, combinatorial optimization

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.29

1 Introduction

Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E, and let $n := |V|$. Given G as input, the *Maximum Stable Set* (MSS) problem calls for the computation of the size of the largest stable set of G (a subset of V with no pair of vertices sharing an edge). Letting $STAB(G)$ be the set of all characteristic vectors of stable sets in G, i.e., of binary vectors $x \in \{0,1\}^n$ where, given a stable set $S \subseteq V$ and for all nodes $i \in V$, $x_i = 1$ if and only if $i \in S$, solving MSS boils down to computing $\alpha(G) := \max \left\{\sum_{i \in V} x_i : x \in STAB(G)\right\}$, where $\alpha(G)$ is the so-called *stability number* of the graph.

MSS is one of Karp's 21 $\mathcal{NP}$-hard problems [15] and it cannot be approximated in polynomial time to within $O(n^{1-\epsilon})$ for any $\epsilon > 0$ unless $\mathcal{P} = \mathcal{NP}$ [14]. To date, it is, arguably, among the most challenging "fundamental" problems in combinatorial optimization to tackle with integer programming techniques.

Introduced by Chvàtal in [6], *Rank Inequalities* (RIs) prescribe that, for any subgraph $G[U]$ induced by $U \subseteq V$, at most $\alpha(G[U])$ of its vertices can be part of a stable set of G:

© Stefano Coniglio and Stefano Gualandi;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 29; pp. 29:1–29:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Definition 1.** The set of all RIs is: $\sum_{i \in U} x_i \leq \alpha(G[U])$, for all $U \subseteq V$.

From a combinatorial perspective, RIs are all the inequalities with binary left-hand side (LHS) coefficients which are valid for $STAB(G)$.[1] These inequalities are very general, as many families of valid inequalities known for $STAB(G)$ are obtained as a special case of RIs when restricting the induced subgraph $G[U]$ to specific topologies (such as cliques, holes, wheels, webs, and antiwebs).

In this work, we propose a novel approach for the separation of RIs where, rather than imposing topological restrictions on the induced subgraph $G[U]$, we assume the right-hand side (RHS) of the inequalities to be fixed to a given (but arbitrary) constant.

To our knowledge, the only methodology that has been developed to separate RIs without topological restrictions is the one proposed [19], which relies on the *edge projection* operator introduced in [16]. Although the method in [19] allows for the generation of RIs without a specific topological restriction, it is heuristic in nature and it can halt before all the violated RIs have been found. See [18] for a study on the impact of those and other (heuristically separated) cuts when solving MSS via branch-and-cut. Recent work on integer programming methods for MSS, partially belonging to the same stream of works, can be found in [9, 10].

The paper is organized as follows. In Section 2, after discussing on the nature of RIs and their separation problem, topology-free RIs with a given RHS are introduced. Our method for their separation is described in Section 3, where we also investigate the polyhedral nature of the corresponding separation problem. Section 4 outlines the main algorithmical aspects of our techniques. Computational results are reported and illustrated in Section 5, while Section 6 draws some concluding remarks.

2 Rank inequalities and topology-free rank inequalities with a fixed right-hand side

Let $RSTAB(G) := \left\{x \in \mathbb{R}^n_+ : \sum_{i \in U} x_i \leq \alpha(G[U]), \forall U \subseteq V\right\}$ be the *closure* of RIs. As it is easy to see, optimizing over this set is, as for MSS, both $\mathcal{NP}$-hard and inapproximable in polynomial time to within $O(n^{1-\epsilon})$ for any $\epsilon > 0$. This is because, for $U = V$, the set of RIs contains the inequality $\sum_{i \in V} x_i \leq \alpha(G)$, whose sole introduction into any relaxation of MSS suffices to obtain $\alpha(G)$.

Due to the equivalence between optimization and separation established in [12], it follows that, given a point x^*, the separation problem of RIs, calling for a subset U of vertices such that $\sum_{i \in U} x^*_i > \alpha(G[U])$, or for a proof that no such subset exists, is also $\mathcal{NP}$-hard.

In integer programming, the $\mathcal{NP}$-hardness of a separation problem is, usually, not an issue *per se*.[2] There are many cases of computationally affordable algorithms in which $\mathcal{NP}$-hard separation problems are routinely solved, often by solving an instance of the very optimization problem being tackled, albeit of smaller size. See, for instance, the pioneering work in [8].

In the context of RIs, the situation is even worse. Indeed, not only separating a RI is $\mathcal{NP}$-hard, but even verifying whether a given inequality is a RI is a difficult problem. First, let us define the decision version of MSS (MSS-d) which, given an integer L, asks whether G contains a stable set of size $\geq L$, i.e., whether $\alpha(G) \geq L$. The following holds:

[1] Indeed, $\pi x \leq \pi_0$ is valid for some $P \subseteq \mathbb{R}^n$ if and only if $\pi_0 \geq \max\{\pi x : x \in P\}$. When restricting to π to $\{0,1\}^n$ and $P = STAB(G)$, the definition of RIs follows.

[2] Due to the equivalence between optimization and separation, an $\mathcal{NP}$-hard optimization problem always has at least one family of valid inequalities which is $\mathcal{NP}$-hard to separate.

▶ **Observation 2.** *Given a graph $G = (V, E)$ and a vector $(\pi, \pi_0) \in \mathbb{R}^{n+1}$, it it strongly $\mathcal{NP}$-hard to decide whether $\pi x \leq \pi_0$ is a RI.*

Proof. We can easily establish a Cook-reduction from MSS-d (with input L and G) to the problem of verifying whether $\pi x \leq \pi_0$ is a RI. Indeed, it suffices to call, for all $\pi_0 \in \{L, \dots, n\}$ (thus, $n - L + 1$ times), a routine which solves the problem of membership to the class of RIs with input G and the inequality $\pi x \leq \pi_0$, with $\pi_i = 1$ for all $i \in V$. Since, for the given π, $\pi x \leq \pi_0$ is a RI if and only if $\pi_0 = \alpha(G)$, as soon as the routine returns answer YES for some π_0, we conclude $\pi_0 = \alpha(G)$, thus providing answer YES to MSS-d. If the membership routine returns answer NO for all values of π_0, we conclude that MSS-d has answer NO. ◀

From a cutting plane perspective, especially when cut generation is embedded within a branch-and-cut algorithm, one would arguably look for a small number of inequalities which, jointly, yield the largest bound improvement over the initial relaxation (see [7] for a cutting plane algorithm designed to achieve this via bilevel programming, and [1, 2] for a method which employs cut diversity). With RIs, as we mentioned, the *single* inequality $\sum_{i \in V} x_i \leq \alpha(G)$ always suffices to bring the bound obtained with *any* relaxation of MSS down to $\alpha(G)$. It is thus clear that, if we aim at a practical method relying on the separation of RIs within an efficient algorithm, some restrictions must be introduced.

The restriction that we consider in this paper is not a topological one. Rather, we investigate the problem of separating RIs when their RHS is fixed to an (arbitrary, small) constant $k \in \mathbb{N}$. We refer to such set of RIs as RI_ks.

▶ **Definition 3.** The set of all RI_ks is: $\sum_{i \in U} x_i \leq k$, for all $U \subseteq V : \alpha(G[U]) = k$.

Note that we can optimize over $RSTAB(G)$ by separating RI_ks for all values of $k \in \{1, \dots, n\}$, a feature which cannot be achieved with traditional approaches where topological restrictions are introduced. The assumption on a small k with (in particular) $k \ll \alpha(G)$ is made so as to arrive at a separation problem which is not too hard to solve in practice, as we will better see in the next sections.

From a combinatorial perspective, the following holds:

▶ **Observation 4.** *For any given $k \in \mathbb{N}$, the LHS of a RI_k is the incidence vector of a subgraph $G[U]$ with a K_{k+1}-free complement.*

Proof. By definition, $\sum_{i \in U} x_i \leq k$ is a RI_k if and only if $k = \alpha(G[U])$. If the complement of $G[U]$ is not K_{k+1}-free, then $G[U]$ contains $k+1$ completely disconnected vertices. Thus $\alpha(G[U]) \geq k + 1 > k$ and $\sum_{i \in U} x_i \leq k$ is not a RI_k. ◀

The observation shows that, given any RI_k with vertex set U, $G[U]$ has a K_2-free complement for $k = 1$, a K_3-free complement for $k = 2$, a K_4-free complement for $k = 3$, and so on. See Figure 1 for an illustration.

3 Separation of topology-free rank inequalities

For a given $k \in \mathbb{N}$, let $RSTAB_k(G)$ be the closure of all RI_ks, i.e., of all RIs with a RHS equal to k. Throughout the paper, our aim is:

▶ **Aim.** *Given a (reasonably small) upper bound $\bar{k}$ on k, optimize over $\bigcap_{k=1}^{\bar{k}} RSTAB_k(G)$.*

The idea is of investigating the tightness of the bound given by $\bigcap_{k=1}^{\bar{k}} RSTAB_k(G)$ within a pure cutting plane method which, at each iteration, looks for a violated RI_k for each

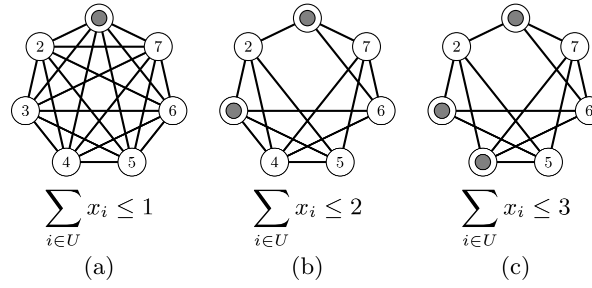


Figure 1 Three induced subgraphs $G[U]$ of the complete graph $G = K_8$ with three RI_ks with $k = 1, 2, 3$, with the corresponding maximum stable set highlighted in gray: (a) an induced subgraph with a K_2-free complement (a clique) and $\alpha(G[U]) = 1$, (b) an induced subgraph with a K_3-free complement and $\alpha(G[U]) = 2$, (c) an induced subgraph with a K_4-free complement and $\alpha(G[U]) = 3$.

$k \in \{1, \ldots, \bar{k}\}$. The overall goal is of assessing whether, even with small values of $\bar{k}$, the bound provided by $\bigcap_{k=1}^{\bar{k}} RSTAB_k(G)$ is stronger than that obtained by computing Lovász's Theta function with semidefinite programming.

3.1 Different separation problems

Given $k \in \mathbb{N}$, the separation problem (in its optimization version) of RI_ks corresponds to the following combinatorial optimization problem:

▶ **Problem 1** (Maximum Weighted Subgraph with Given Stability Number (MWS-GSN)). *Given a graph $G = (V, E)$, a weight vector $x^* \in \mathbb{R}^n$, and an integer k, find a subset of vertices $U \subseteq V$ of maximum weight inducing a subgraph $G[U]$ with stability number equal to k.*

The restriction of RIs to a given RHS does not yield an easier separation problem, at least not from a theoretical perspective:

▶ **Observation 5.** *MWS-GSN is strongly $\mathcal{NP}$-hard.*

Proof. Consider the decision version of MWS-GSN, which asks whether G contains an induced subgraph $G[U]$ of weight $\geq M$ and $\alpha(G[U]) = k$. Letting $M = 0$ and $x^* \in \mathbb{R}^n_+$, the problem has answer YES/NO if and only if MSS-d with input $L = k$ and G has answer YES/NO. ◀

For computational ease but without loss of generality, we introduce an alternative way to optimize over $\bigcap_{k=1}^{\bar{k}} RSTAB_k(G)$, which only requires to solve a relaxation of MWS-GSN. Consider the following inequalities, which we refer to as $\text{RI}_k^{\leq}$s:

▶ **Definition 6.** The set of all $\text{RI}_k^{\leq}$s is: $\sum_{i\in U} x_i \leq k$, for all $U \subseteq V : \alpha(G[U]) \leq k$.

The relationship between RI_ks and $\text{RI}_k^{\leq}$s is as follows:

▶ **Proposition 7.** *For any $k \in \mathbb{N}$, a $RI_k^{\leq}$ is either a RI_k or it is dominated by a $RI_{k'}^{\leq}$ for some $k' < k$.*

Proof. Let $\sum_{i\in U} x_i \leq k$ be a $\text{RI}_k^{\leq}$. If $\alpha(G[U]) = k$, it is a RI_k. If $\alpha(G[U]) < k$, the inequality $\sum_{i\in U} x_i \leq k'$ is a $\text{RI}_{k'}$ with $k' = \alpha(G[U]) < k$. It also dominates $\sum_{i\in U} x_i \leq k$: the two inequalities have the same left-hand side, while the second one has a strictly smaller right-hand side. ◀

Proposition 7, when applied recursively, implies that, by iteratively separating $\mathrm{RI}_k^{\leq}$s *in lieu* of RI_ks for increasing values of $k \in \{1, \ldots, \bar{k}\}$, the only inequalities that will be generated are RI_ks, thus showing that the adoption of $\mathrm{RI}_k^{\leq}$ is without loss of generality.

The separation problem (in optimization version) for $\mathrm{RI}_k^{\leq}$s is:

▶ **Problem 2** (Maximum Weighted Subgraph with Bounded Stability Number (MWS-BSN))**.** *Given a graph $G = (V, E)$, a weight vector $x^* \in \mathbb{R}^n$, and an integer k, find a subset of vertices $U \subseteq V$ of maximum weight inducing a subgraph $G[U]$ with stability number smaller than or equal to k.*

Observe that MWS-BSN is a relaxation of MWS-GSN.

3.2 MWS-BSN: the separation problem of $\mathrm{RI}_k^{\leq}$s

We will now investigate the separation problem for $\mathrm{RI}_k^{\leq}$s: MWS-BSN. Previous work on a closely related problem can be found in [3, 4].

The aim of this section is to show how MWS-BSN can be solved via branch-and-cut. For the purpose, we will introduce a set of inequalities which are necessary to correctly formulate it in the vertex-space. We remark that those inequalities are valid for MWS-BSN only, and *not* for MSS.

Observe that, for any $U \subseteq V$, $\alpha(G[U]) \leq k$ if and only if, for all stable sets S of G with $|S| = k + 1$, $|S \cap U| \leq k$. We deduce that, letting $u \in \{0, 1\}^n$ be the characteristic vector of U, the following constraints are both necessary and sufficient for u to be a feasible solution to MWS-BSN. We refer to them as *Cover Inequalities* (CIs) (as they play the same role as cover inequalities for the 0-1 knapsack problem):

▶ **Definition 8.** Let $\mathcal{S}^{=k+1}$ be the collection of all stable sets of G of cardinality equal to $k + 1$. The set of CIs is: $\sum_{i \in S} u_i \leq k$, for all $S \in \mathcal{S}^{=k+1}$.

Note that, as much as RI_ks for $k > 1$ can be seen as a generalization of clique inequalities, CIs can be regarded as a generalization of edge inequalities which, in the separation problem of clique inequalities (the max clique problem), prevent the presence of stable sets of size 2 in the induced subgraph.

From a polyhedral perspective, the following holds:

▶ **Proposition 9.** *CIs are not facet defining for MWS-BSN.*

Proof. Consider a CI $\sum_{i \in S} u_i \leq k$. If S is not an inclusion-wise maximal stable set, there is a larger stable set S' containing it. It follows that the inequality $\sum_{i \in S'} u_i \leq k$ dominates $\sum_{i \in S} u_i \leq k$, as it is obtained from the latter by lifting each variable u_j with $j \in S' \setminus S$ with a unit coefficient. ◀

Consider now the following constraints, which we call *Lifted Cover Inequalities* (LCIs):

▶ **Definition 10.** Let $\mathcal{S}_M^{\geq k+1}$ be the collection of *maximal* stable sets of G of cardinality $\geq k + 1$. The set of LCIs is: $\sum_{i \in S} u_i \leq k$, for all $S \in \mathcal{S}_M^{\geq k+1}$.

LCIs can be shown to be facet defining for MWS-BSN. For the purpose, we first introduce the following lemma:

▶ **Lemma 11.** *Let $S \in \mathcal{S}_M^{\geq k+1}$. LCIs are facet defining for MWS-BSN when restricted to $G[S]$, i.e., to the subspace where $u_i = 0$ for all $i \in V \setminus S$.*

Proof. Since $G[S]$ is a stable set, any subset $S' \subseteq S$ of at most k vertices yields a feasible solution to MWS-BSN. The convex hull of such solutions is thus given by three groups of constraints: $\sum_{i \in S} u_i \leq k$; $u_i \geq 0$ for all $i \in S$; and $u_i \leq 1$ for all $i \in S$. Together, they form a totally unimodular system. Since, by definition of LCIs, $|S| \geq k+1$, the inequality $\sum_{i \in S} u_i \leq k$ is not implied nor dominated by any of the constraints in the other two groups and, thus, it is facet defining. ◀

The following can now be established:

▶ **Theorem 12.** *LCIs are facet defining for MWS-BSN.*

Proof. Let $j_1, \ldots, j_{|V \setminus S|}$ be an ordering of $V \setminus S$. Let M be the set of integer solutions to MWS-BSN and let M^ℓ be the subset of M restricted to $u_{j_k} = 0$ for all $k \in \{\ell+1, \ldots, |V \setminus S|\}$, where $\{\ell+1, \ldots, |V \setminus S|\}$ is considered equal to $\emptyset$ if $\ell + 1 > |V \setminus S|$. We employ a sequential lifting argument. Starting from the inequality $\sum_{i \in S} u_i \leq k$ which, as of Lemma 11, is facet defining for $\text{conv}(M^0)$, at each lifting iteration ℓ we obtain a facet of $\text{conv}(M^\ell)$ and, for $\ell = |V \setminus S|$, a facet of $\text{conv}(M)$.

At iteration ℓ, given the lifted inequality $\sum_{i \in S} u_i + \sum_{k \in \{1,\ldots,\ell-1\}} \lambda_{j_k} u_{j_k} \leq k$, valid for $\text{conv}(M^{\ell-1})$ for some $\lambda_{j_1}, \ldots, \lambda_{j_{\ell-1}} \in \mathbb{R}^+$, we compute the (largest) coefficient λ_{j_ℓ} for which the new inequality $\sum_{i \in S} u_i + \sum_{k \in \{1,\ldots,\ell-1\}} \lambda_{j_k} u_{j_k} + \lambda_{j_\ell} u_{j_\ell} \leq k$ is valid for $\text{conv}(M^\ell \cap \{u_{j_\ell} = 1\})$ (and thus for $\text{conv}(M^\ell)$). This lifting problem reads:

$$\Lambda_\ell = \max_{u \in \{0,1\}^n} \sum_{i \in S} u_i + \sum_{k \in \{1,\ldots,\ell-1\}} \lambda_{j_k} u_{j_k} \tag{1a}$$

$$\text{s.t. } u_{j_\ell} = 1 \tag{1b}$$

$$u_{j_k} = 0 \;\; \forall k \in \{\ell+1, \ldots, |V \setminus S|\} \tag{1c}$$

$$\alpha(G[\{i \in V : u_i = 1\}]) \leq k. \tag{1d}$$

Since S is maximal by definition of LCIs and $j_\ell \notin S$, $\exists i \in S : \{i, j_\ell\} \in E$. Let then S' be a subset of S containing vertex i, of cardinality $|S'| = k$. Since S' is a stable set and $\{i, j_\ell\} \in E$, $\alpha(G[S' \cup \{j_\ell\}]) = \alpha(G[S']) = |S'| = k$. By letting $u_{j_\ell} = 1$ and $u_i = 1$ for all $i \in S'$ we thus obtain a feasible solution to the lifting problem of value k. This shows that $\Lambda_\ell \geq k$. Since the lifted inequality is valid if and only if $\Lambda_\ell + \lambda_{j_\ell} \leq k$, we deduce $\lambda_{j_\ell} \leq 0$.

To show that $\lambda_{j_\ell} = 0$ for all $\ell \in \{1, \ldots, |V \setminus S|\}$, first note that, if $\lambda_{j_k} = 0$ for all $k \in \{1, \ldots, \ell-1\}$, then $\Lambda_\ell \leq k$. Due to the previous argument, this implies $\Lambda_\ell = k$ and, hence, $\lambda_{j_\ell} = 0$. Also note that, for $\ell = 1$, no terms $\lambda_{j_k} u_{j_k}$ appear in the objective function and, hence, $\lambda_{j_1} = 0$. The claim then follows by induction (if $\lambda_{j_1}, \ldots, \lambda_{j_{\ell-1}} = 0$, then $\lambda_{j_\ell} = 0$), proving that, at the end of the lifting procedure, any LCI is lifted back to itself, and, therefore, is facet defining. ◀

Letting $u^* \in [0,1]^n$ (corresponding to a, possibly infeasible, solution to MWS-BSN), the separation problem for LCIs (in search version) reads:

▶ **Problem 3** (SEParation problem for LCIs (LCI-SEP))**.** *Given a graph $G = (V, E)$, a vector of vertex weights $u^* \in \mathbb{R}^n$, and an integer k, find a maximal stable set S of G with both weight and cardinality greater than or equal to $k+1$, or prove that none exists.*

Not surprisingly, the following holds:

▶ **Proposition 13.** *LCI-SEP is $\mathcal{NP}$-hard.*

Algorithm 1: Exact algorithm for the optimization over $\bigcap_{k=1}^{\bar{k}} RSTAB_k(G)$.

Solve the (current) relaxation of MSS; let x^* be its solution;
Let $k := 1$;
while $k \leq \bar{k}$ **do**
  solve MWS-BSN via branch-and-cut, separating LCIs;
  if *the corresponding* $RI_k^{\leq}$ *is violated* **then**
    add it to the relaxation of MSS;
    let $k := 1$;
  else
    let $k := k + 1$;
  end
end

Proof. MSS-d with input L and G has answer YES if and only if LCI-SEP with input $k = L - 1$ admits a feasible solution. ◀

Note that, due to the equivalence between optimization and separation [11], the facet-definingness of LCIs and their $\mathcal{NP}$-hardness imply, *en passant*, the $\mathcal{NP}$-hardness of MWS-BSN.

We remark that, since CIs are necessary to formulate MWS-BSN in the vertex space and there is an exponential number of them, solving MWS-BSN in that space via branch-and-bound requires a cut generation procedure.

4 Algorithmic aspects

In this section, we provide an outline of our algorithm for optimizing over $\bigcap_{k=1}^{\bar{k}} RSTAB_k(G)$ and then discuss a few of its aspects.

4.1 Algorithm outline

The overall algorithm by which the function $\sum_{i \in V} x_i$ is maximized over $RSTAB_k(G)$ can be summarized as follows:

4.2 Domination aspects of RIs: connectedness of $G[U]$

An easy condition under which a RI is dominated is the following one:

▶ **Observation 14.** *Any RI corresponding to a disconnected $G[U]$ is dominated.*

Proof. Assuming that $G[U]$ contains ℓ connected components $G[U_1], \dots, G[U_\ell]$, $\alpha(G[U]) = \sum_{j=1}^{\ell} \alpha(G[U_\ell])$. Hence, $\sum_{i \in U} x_i \leq \alpha(G[U])$ is the linear combination with unit weights of the ℓ inequalities $\sum_{i \in U_j} x_i \leq \alpha(G[U_j])$, for $j \in \{1, \dots, \ell\}$. ◀

To prevent the introduction of RI_ks with a disconnected $G[U]$, we identify (in linear time) the connected components $G[U_1], \dots, G[U_k]$ of $G[U]$ after each RI_k is generated. We then introduce a RI for each component, *in lieu* of the original one. For that, we recompute the RHS of each new inequality as $\alpha(G[U_j])$ (which is an easy task, provided that $|U|$ is reasonably small). Note that, since, for all $j \in \{1, \dots, k\}$, $\alpha(G[U_j]) \leq \alpha(G[U]) = k$, all the

inequalities obtained after the decomposition of $G[U]$ are $\mathrm{RI}_{k'}$s with $k' < k$, thus being in $\bigcap_{k=1}^{\bar{k}} RSTAB_k(G)$.

4.3 Practical separation of LCIs

First, we note that, in the context of a branch-and-cut algorithm for MWS-BSN, LCIs can be separated on the *incumbent* solution. This allows to consider only the case where u is a binary vector. If this is the case, LCI-SEP becomes *exactly* an instance of MSS-d with $L = k+1$ due to the weight of the stable set becoming equal to its cardinality. Note also that, conveniently, LCIs can be obtained by separating CIs and, then, making the corresponding stable set maximal *a posteriori* via a greedy algorithm, in $O(n^2)$.

We remark that the separation of $\mathrm{RI}_k^{\leq}$s for the MSS problem entails, via the separation of CIs/LCIs, the solution of, yet again, MSS. Two things must be noted though: 1) the separation problem for CIs/LCIs can be solved on the subgraph induced by the incumbent solution u of MWS-BSN, which is much smaller, in practice, than G; 2) assuming $k \ll \alpha(G)$ for a sufficiently small k, finding a stable set of size k is, in practice, a computationally more affordable task than computing $\alpha(G)$.

In our computations, we will carry out the separation of CIs/LCIs with the exact solver Cliquer [17], which implements a combinatorial branch-and-bound algorithm not relying on mathematical programming relaxations.

4.4 Separating $\mathrm{RI}_k^{\leq}$s on the support of x^*

We will restrict ourselves to the subgraph induced by the solution vector being separated, x^* in this case, also when solving MWS-BSN. For this problem, a simple argument also allows to fix $u_i = 0$ for all $i \in V$ where $x_i^* = 1$. This is because, if $x_i^* = 1$, assuming that the LP relaxation of MSS contains, at least, all edge inequalities (which is always the case in our implementation), we have that, for all $j \in V : \{i, j\} \in E, x_j^* = 0$. As a consequence, when the aforementioned restriction is in place, vertex i is isolated. Since we are looking for inequalities where $G[U]$ is connected, node i can thus be safely discarded.

4.5 Heuristic procedure

To speedup the cutting plane algorithm for RI_ks, we also introduce a simple greedy heuristic for their separation. After sorting the vertices of V in nonincreasing order of x^*, we add them to U one at a time, until a maximal clique is formed (this way, only stable sets of cardinality 1 are introduced). Then, we add, in the previously found order, the next $k-1$ nodes. After this operation, the stability number of $G[U]$ is, at most, k. Then, for each vertex currently not in of U, we add it to U only if it does not form a stable set of cardinality $k+1$. If it does, we skip it and continue to the next vertex.

The algorithm runs in $O(n \log n + n^{k+1})$, where $O(n \log n)$ accounts for sorting and $O(n^k)$ is the number of operations needed to check whether a new vertex increases the stability number of the current subgraph past the upper bound of k. The latter operations are executed $O(n)$ times. Note that, by construction, any solution found by this heuristic is maximal. If, after the exploration of a given amount of nodes, the heuristic terminates without finding a violated inequality (the amount is set to 2 millions in our experiments), we resort to branch-and-cut.

5 Computational study

We now report on a set of results obtained with the algorithm that we described in the previous sections for the separation of topology-free RIs with a given RHS.

We remark that computational efficiency is not our primary concern here. Rather, we focus on assessing the *quality of the bounds* obtained with $\bigcap_{k=1}^{\bar{k}} RSTAB_k(G)$ for increasing values of $\bar{k}$. We will compare those bounds to those obtained when optimizing over $QSTAB(G)$ (the relaxation containing all clique inequalities) and when employing Lovász's Theta function $\vartheta(G)$, which yields one of the tightest upper bounds to MSS known in the literature (always at least as tight as that obtained with $QSTAB(G)$). We refer to the latter two bounds as $\alpha_{QSTAB}(G)$ and $\alpha_{\vartheta}(G)$. Throughout our experiments, we adopt $QSTAB(G)$ as the initial relaxation of MSS. Given an upper bound UB, we will measure its quality in terms of the fraction of gap that it closes w.r.t. $\alpha_{QSTAB}(G)$. Formally, we define the *closed gap* as:

$$\text{Closed Gap }\% := \left(1 - \frac{UB - \alpha(G)}{\alpha_{QSTAB}(G) - \alpha(G)}\right) 100.$$

5.1 Instances

We consider three groups of instances, all corresponding to sparse graphs (we recall that sparse graphs are usually much harder to solve than dense ones):

1. The first group contains uniform random graphs, generated with `rudy` [20]. They have 60, 70, and 80 vertices and an edge density between 5% and 25%. Those instances are particularly useful to measure the impact of $\text{RIs}_{\bar{k}}^{\leq}$ with $\bar{k} > 3$.
2. The second group is a subset of the largest instances among those used in [13] to solve MSS via SDP techniques. They are very sparse, with a density between 1% and 5%.
3. The third group is a small subset of sparse graphs taken from the DIMACS challenge on the max clique problem. All the instances for which either $\alpha_{QSTAB}(G) = \alpha(G)$ or for which $\alpha_{QSTAB}(G)$ cannot be computed *exactly* within the time limit are discarded.

5.2 Implementation details

Our algorithm is coded in C, using Gurobi 7.0 as MILP solver. We adopt the `parallel` setting, with 8 threads and default parameters. In all the separation problems, we set `solutionlimit=1`, imposing a violation cutoff of 0.01. For the separation of LCIs, we use Cliquer 1.21. The value $\vartheta(G)$ is obtained with DSDP 5.8. All the results are produced within a time limit of 7200 seconds (two hours) on an Intel i7-3770 CPU @ 3.40GHz desktop computer with 8 cores, with 16GB RAM.

5.3 A small example: the Chvàtal graph

As an illustrative example, we report the results obtained over the Chvàtal graph, the smallest triangle free 4-colorable 4-regular graph, see [5].

Figure 2 shows 11 RIs with $\bar{k} = 3$ generated by our topology free cutting plane algorithm, assuming $QSTAB(G)$ as the initial relaxation. Apart from the fourth inequality, which is isomorphic to the web inequality $W(8,3)$, none of the remaining RIs corresponds to any of the valid inequalities with a given topology that are known in the literature. While the bound obtained with $QSTAB(G)$ is $\alpha_{QSTAB}(G) = 6$ (corresponding to the solution $x_i = \frac{1}{2}$ for all $i \in V$) and that obtained with Lovász's Theta function is $\alpha_{\vartheta}(G) = 4.895$, with RIs_ks and $k = 3$ we obtain a better bound equal to 4.5.

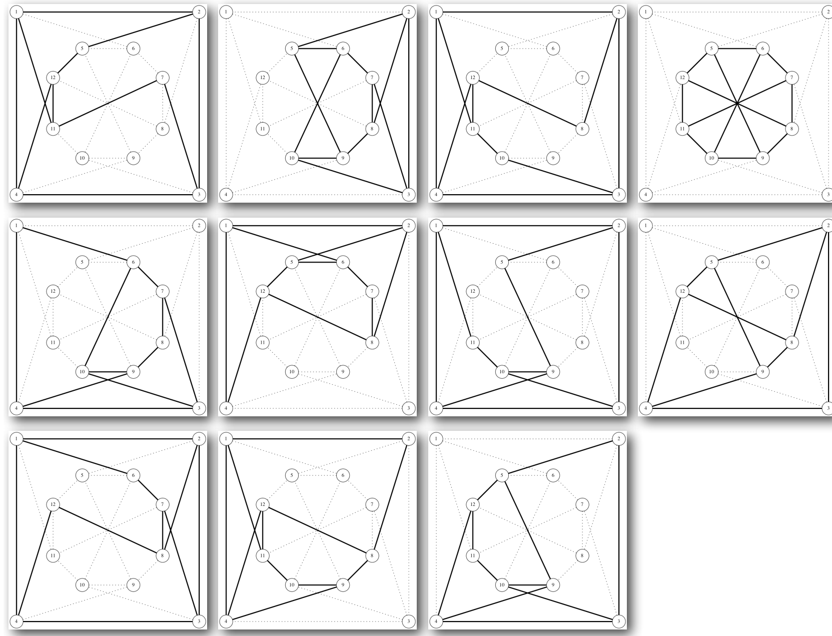


Figure 2 The set of the 11 RI_ks which are obtained when optimizing over $RSTAB_k(G)$ with $k = 3$ on the Chvàtal graph. They yield a bound of 4.5, as opposed to $\alpha_{QSTAB}(G) = 6$ and $\vartheta(G) = 4.895$.

5.4 Computational Results

Figure 3 reports the percentage of closed gap plotted against the running time for instance `r-70-10` in group 1, obtained when executing Algorithm 1 with $\bar{k} = 5$. This plot clearly shows that, in the very first iterations of the algorithm, RIs are already able to close a large percentage of the gap, closing 90% of it in only 40 seconds. After 250 seconds, nearly 100% of the gap is closed. The additional 250 seconds are only necessary to prove (computationally) that the upper bound that has been obtained cannot be improved any further. The plot in the figure illustrates a behaviour which can be observed in all the results that we will discuss in the next paragraph.

The results obtained on the three groups of instances are summarized in Table 1. For each value of $\bar{k} = \{2, 3, 4, 5\}$, we report the Upper Bound (UB) that has been found, the running time in seconds (Time), and the number of cuts that were generated (Cuts). We also report the average closed gap (Avg ClGap), as computed over the instances belonging to each group.

On the first group of instances, our algorithm manages to close, on average, more than 50% of the open gap already with $\bar{k} = 2$. Larger values of $\bar{k}$ yield a larger closed gap, up to more than 80% with $\bar{k} = 5$. Note though that this result is counterbalanced by an increase of running time as, for $\bar{k} = 5$, most instances hit the time limit of 2 hours.

We remark that, in the first two groups of instances, RI_ks with $\bar{k} = 3$ suffice to obtain stronger bounds than those achieved with Lovász's Theta function $\vartheta(G)$. On the instances in group 1 we register, on average, 67.7% of gap closed with RI_k, as opposed to 67.6% with $\vartheta(G)$, a value which increases to 73.2% for group 2 as opposed, for that group, to the 65.8% obtained with $\vartheta(G)$.

The improvement w.r.t. $\vartheta(G)$ further increases when considering $\bar{k} = 4$ and $\bar{k} = 5$. The quality of the bound improvement becomes hard to assess though on the third group of

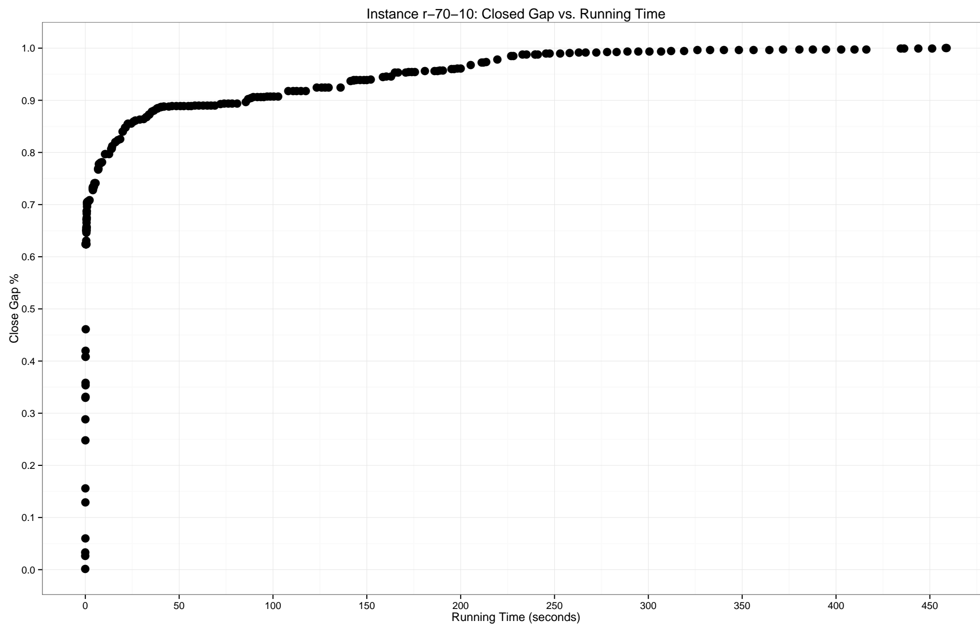


Figure 3 Percentage of closed gap plotted against the running time for instance `r-70-10` in group 1, obtained when executing Algorithm 1 with $\bar{k} = 5$.

instances where, already with $\bar{k} = 2$, our algorithm hits the time limit in three cases out of seven.

We remark that the cuts that we generate are quite sparse. As an example, consider instance `r-70-10` from group 1 (containing 70 nodes). On average, we generate inequalities with $|U|$ (corresponding to the number of nonzeros in the LHS) equal to 5.2 for $k = 2$, 8.01 for $k = 3$, 10.7 for $k = 4$, and 12.9 for $k = 5$.

To conclude, we highlight the results on the instance `hamming6-4`: with only 11 cuts with $\bar{k} = 2$, generated on top of those in $QSTAB(G)$, our algorithm achieves the optimal bound equal to 4, while both $QSTAB(G)$ and $\vartheta(G)$ yield a larger upper bound equal to 5.33.

6 Concluding remarks

We have addressed the separation of topology-free rank inequalities with a fixed (arbitrary) right-hand side (RI$_k$s). We have proposed a methodology to optimize over the closure of RI$_k$s for all $k \in \{1, \dots, \bar{k}\}$, investigating the arising separation problem and its polyhedral structure. For its solution, we have proposed a branch-and-cut method which separates facet defining inequalities belonging to an exponentially large family of inequalities that are needed to correctly model the problem.

Overall, RI$_k$s with a small right-hand side $k \ll \alpha(G)$ yield a substantial bound improvement over the bound provided by the fractional clique polytope $QSTAB(G)$. In a number of cases, such bound is also tighter than $\vartheta(G)$, the bound obtained with Lovász's Theta function via semidefinite programming.

Future work includes the development of *ad hoc* algorithms for the separation of RI$_k$s with a small right-hand side k. Due to the bound improvement that, in our experiments, RIs have shown to yield, the effectiveness of such algorithms could allow to add RI$_k$s with $k = 2$ and $k = 3$ to the set of cutting planes that are routinely generated to solve the maximum stable set problem to optimality.

Table 1 Bounds (UB) obtained with RI_ks with $k \in \{2,3,4,5\}$, compared to $\alpha(G)$, $\alpha_{QSTAB}(G)$, and $\vartheta(G)$. Computing times in seconds (Time) and total number of generated cutting planes (Cuts) are also reported. Bounds which are tighter than those obtained with $\vartheta(G)$ are highlighted in bold. Closed Gap, averaged in geometric mean (Avg ClGap), is reported for the three classes of instances also considering instances for which the time limit is met: small uniform random graphs generated with `rudy`, large uniform random graphs taken from [13], and structured instances from the DIMACS challenge.

				RI_k with $\bar{k}=2$			**RI_k with $\bar{k}=\{2,3\}$**			**RI_k with $\bar{k}=\{2,3,4\}$**			**RI_k with $\bar{k}=\{2,3,4,5\}$**		
	$\alpha(G)$	$\alpha_Q(G)$	$\vartheta(G)$	**Cuts**	**UB**	**Time**	**Cuts**	**UB**	**Time**	**Cuts**	**UB**	**Time**	**Cuts**	**UB**	**Time**
r-60-5	31	31.50	31.07	5	**31.00**	0	5	**31.00**	0	5	**31.00**	0	5	**31.00**	0
r-60-10	23	25.63	23.67	42	23.47	1	96	23.14	28	124	**23.00**	77	124	**23.00**	77
r-60-15	18	21.20	19.54	70	19.70	2	189	19.19	77	315	18.97	521	455	**18.81**	2090
r-60-20	7	9.25	7.5	437	8.15	1254	946	7.79	tlim	946	7.79	tlim	947	**7.47**	tlim
r-60-25	14	16.50	14.67	112	15.24	12	348	14.70	413	569	14.38	2814	756	**14.13**	tlim
r-70-5	35	36.00	35.53	5	35.50	0	6	35.50	0	6	35.50	0	8	**35.00**	0
r-70-10	26	28.66	26.86	63	26.78	2	139	26.29	72	230	26.01	417	236	**26.00**	459
r-70-15	21	23.82	21.91	103	22.16	8	265	21.63	289	423	21.37	2199	550	**21.22**	tlim
r-70-20	17	20.52	18.23	119	19.18	30	337	18.46	985	510	18.16	tlim	510	**18.15**	tlim
r-70-25	14	18.13	15.72	144	16.66	39	421	16.04	2039	607	15.71	tlim	607	**15.64**	tlim
r-80-5	39	39.50	39.02	3	**39.00**	0	3	**39.00**	0	3	**39.00**	0	3	**39.00**	0
r-80-10	27	30.50	28.55	69	29.02	8	194	28.38	399	350	27.95	4220	399	**27.76**	tlim
r-80-15	22	26.74	23.65	120	24.76	26	328	23.97	1874	448	23.67	tlim	448	**23.59**	tlim
r-80-20	18	22.78	20.05	145	21.06	41	422	20.40	3335	512	20.22	tlim	512	**20.07**	tlim
r-80-25	16	19.85	17.07	178	18.19	85	477	17.61	tlim	478	17.61	tlim	478	**17.55**	tlim
Avg ClGap			**67.6%**		**53.0%**			**67.7%**			**73.4%**			**80.3%**	
g150.4	59	67.00	61.8	99	62.09	50	250	60.80	tlim	250	60.80	tlim	250	**60.67**	tlim
g150.5	55	64.00	58.73	152	58.56	72	304	57.75	tlim	304	57.75	tlim	304	**57.67**	tlim
g170.3	71	78.50	73.34	76	73.53	44	181	72.16	6861	182	72.15	7415	182	**72.14**	tlim
g200.2	96	100.00	97.17	21	97.00	11	46	**96.00**	378	49	**96.00**	437	50	**96.00**	439
g200.3	83	94.50	86.52	123	86.61	221	202	85.21	tlim	202	85.21	tlim	202	**85.02**	tlim
g300.2	122	141.00	129.47	144	130.43	861	169	**130.07**	tlim	169	**130.07**	tlim	169	**130.07**	tlim
g350.2	133	161.00	143.43	273	146.11	4996	274	145.99	tlim	274	145.99	tlim	274	**145.87**	tlim
g400.1	191	201.00	194.79	33	195.50	131	60	**193.73**	tlim	60	**193.73**	tlim	60	**193.73**	tlim
Avg ClGap			**65.8%**		**61.6%**			**73.2%**			**73.2%**			**73.3%**	
brock200_1	21	38.02	27.46	267	35.59	tlim	267	35.59	tlim	267	35.59	tlim	267	35.59	tlim
C125.9	34	43.06	37.81	188	39.75	409.2	322	39.20	tlim	320	39.21	tlim	322	39.21	tlim
C250.9	44	71.37	56.24	375	66.05	tlim	375	66.05	tlim	375	66.05	tlim	375	66.05	tlim
hamming6-4	4	5.33	5.33	11	**4.00**	1.5	11	**4.00**	1.5	11	**4.00**	1.5	11	**4.00**	1.5
keller4	11	14.83	14.01	314	**13.80**	tlim	318	**13.80**	tlim	314	**13.80**	tlim	314	**13.80**	tlim
MANN_a9	16	18.00	17.48	1	18.00	0.1	1	18.00	0.7	1	18.00	2.7	1	18.00	13.1
sanr200_0.9	45	59.82	49.27	366	55.14	tlim	366	55.13	tlim	366	55.14	tlim	366	55.14	tlim
Avg ClGap			**26.5%**		**19.5%**			**19.9%**			**19.9%**			**19.9%**	

References

1. Edoardo Amaldi, Stefano Coniglio, and Stefano Gualandi. Improving cutting plane generation with 0-1 inequalities by bi-criteria separation. *Experimental Algorithms*, pages 266–275, 2010.
2. Edoardo Amaldi, Stefano Coniglio, and Stefano Gualandi. Coordinated cutting plane generation via multi-objective separation. *Mathematical Programming*, 143(1-2):87–110, 2014.
3. Chitra Balasubramaniam and Sergiy Butenko. The maximum s-stable cluster problem. In *INFORMS 2015 Annual Meeting*. INFORMS, 2015.
4. Chitra Balasubramaniam and Sergiy Butenko. The maximum s-stable cluster problem. Working paper, 2017.
5. Václav Chvátal. The smallest triangle-free 4-chromatic 4-regular graph. *Journal of Combinatorial Theory*, 9(1):93–94, 1970.
6. Vašek Chvátal. On certain polytopes associated with graphs. *Journal of Combinatorial Theory, Series B*, 18(2):138–154, 1975.
7. Stefano Coniglio and Martin Tieves. On the generation of cutting planes which maximize the bound improvement. In *Experimental Algorithms (14th International Symposium, SEA 2015, Paris, France, June 29 – July 1, 2015, Proceedings)*, volume 9125, pages 97–109. Springer International Publishing, 2015.
8. Harlan Crowder, Ellis L. Johnson, and Manfred Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31(5):803–834, 1983.
9. Monia Giandomenico, Adam N Letchford, Fabrizio Rossi, and Stefano Smriglio. Ellipsoidal relaxations of the stable set problem: theory and algorithms. *SIAM Journal on Optimization*, 25(3):1944–1963, 2015.
10. Monia Giandomenico, Fabrizio Rossi, and Stefano Smriglio. Strong lift-and-project cutting planes for the stable set problem. *Mathematical Programming*, 141(1-2):165–192, 2013.
11. Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.
12. Martin Grötschel, Laszlo Lovász, and Alexander Schrijver. *Geometric algorithms and combinatorial optimization*, volume 2 of *Algorithms and Combinatorics*. Springer-Verlag, 1988.
13. Gerald Gruber and Franz Rendl. Computational experience with stable set relaxations. *SIAM Journal on Optimization*, 13(4):1014–1028, 2003.
14. J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.
15. Richard M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Proceedings of a Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series. Plenum Press, 1972.
16. Carlo Mannino and Antonio Sassano. Edge projection and the maximum cardinality stable set problem. *DIMACS series in discrete mathematics and theoretical computer science*, 26:205–219, 1996.
17. Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1):197–207, 2002.
18. Steffen Rebennack, Marcus Oswald, Dirk Oliver Theis, Hanna Seitz, Gerhard Reinelt, and Panos M. Pardalos. A branch and cut solver for the maximum stable set problem. *Journal of combinatorial optimization*, 21(4):434–457, 2011.
19. Fabrizio Rossi and Stefano Smriglio. A branch-and-cut algorithm for the maximum cardinality stable set problem. *Operations Research Letters*, 28(2):63–74, 2001.
20. Yinyu Ye. Rudy random graph generator. `http://web.stanford.edu/~yyye/yyye/Gset`.

Graph Partitioning with Acyclicity Constraints

Orlando Moreira[1], Merten Popp[2], and Christian Schulz[3]

1 **Intel Corporation, Eindhoven, The Netherlands**
`orlando.moreira@intel.com`
2 **Intel Corporation, Eindhoven, The Netherlands**
`merten.popp@intel.com`
3 **Karlsruhe Institute of Technology, Karlsruhe, Germany; and University of Vienna, Vienna, Austria**
`christian.schulz@{kit.edu, univie.ac.at}`

Abstract

Graphs are widely used to model execution dependencies in applications. In particular, the NP-complete problem of partitioning a graph under constraints receives enormous attention by researchers because of its applicability in multiprocessor scheduling. We identified the additional constraint of acyclic dependencies between blocks when mapping streaming applications to a heterogeneous embedded multiprocessor. Existing algorithms and heuristics do not address this requirement and deliver results that are not applicable for our use-case. In this work, we show that this more constrained version of the graph partitioning problem is NP-complete and present heuristics that achieve a close approximation of the optimal solution found by an exhaustive search for small problem instances and much better scalability for larger instances. In addition, we can show a positive impact on the schedule of a real imaging application that improves communication volume and execution time.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases Graph Partitioning, Computer Vision and Imaging Applications

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.30

1 Practical Motivation

The context of this research is the development of computer vision and imaging applications at Intel Corporation. These applications have high demands for computational power but often need to run on embedded devices with severely limited compute resources and a tight thermal budget. Our target platform is a heterogeneous multiprocessor for advanced imaging and computer vision and is currently used in Intel processors. It is designed for low power and has small local program and data memories. To cope with the memory constraints, the application developer currently has to *manually* break the application, which is given as a directed dataflow graph, into smaller blocks that are executed one after another. The quality of this partitioning has a strong impact on communication volume and performance. However, for large graphs this is a non-trivial task that requires detailed knowledge of the hardware. Hence, the task should be handled by a well-designed algorithm instead.

There are many existing heuristics for partitioning graphs into blocks of nodes of roughly equal size. However, our platform has the requirement that there must not be a cycle in the dependencies between the blocks because they have to be executed one after another.

The contributions of this work are the identification of a new variation of the graph partitioning problem, proofs it is NP-complete and hard to approximate, as well as the implementation and evaluation of heuristics that address this problem. First, we present

© Orlando Moreira, Merten Popp, and Christian Schulz;
licensed under Creative Commons License CC-BY

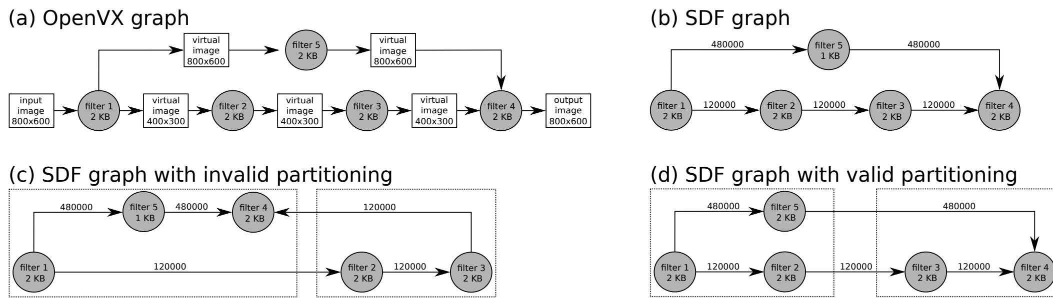


Figure 1 Depiction of an imaging application graph. The nodes represent processing kernels and are annotated with the size of the program binaries. The OpenVX graph as specified by the developer using the API standard is shown in (a), virtual images are intermediate data. The initial SDF representation is shown in (b) where the edges are annotated with the buffer size, (c) shows an invalid partition with minimal edge cut, but a bidirectional connection between blocks and thus a cycle in the quotient graph. A valid partitioning with minimal edge cut is shown in (d).

all necessary background information on the application graph and hardware and explain our additional constraint in Section 2. We then continue to briefly introduce all basic concepts and related work in Section 3. We have not been able to identify works that address the aforementioned constraint of our problem variant, which originates from the hardware platform and disallows cycles between blocks. The proofs are found in Section 4 and the proposed heuristic algorithms in Section 5. We perform a number of experiments in Section 6, where we use small graphs to compare our heuristics against an optimal algorithm that uses exhaustive search. We then evaluate our heuristics for larger graphs in the context of a real-world imaging application and estimate the impact on the application. In addition, we demonstrate the scalability of our heuristics with a set of large graphs. Finally, we conclude in Section 7.

2 Background

Computer vision and imaging applications can often be expressed as stream graphs where nodes represent tasks that process the stream data and edges denote the direction of the dataflow. The widely-accepted industry standard OpenVX [11] released in 2014 by the Khronos group uses a graph-based execution model. With the OpenVX API the developer can specify the data flow of the application as a graph independent of hardware constraints. The hardware vendor on the other hand can provide an API implementation that uses advanced optimizations [21] like specialized hardware, parallelized and pipelined node execution, overlapped computation and data transfers and aggregated data transfers to avoid a round trip to external memory.

The application is specified as a Directed Acyclic Graph (DAG) in OpenVX. The nodes of the DAG are either kernels (small, self-contained functions) or data objects. Edges denote data dependencies and always connect exactly one kernel with one data object. No cycles or feedback loops are allowed [11]. The need of some imaging algorithms to access previous data (e.g. video stabilization) is addressed by special OpenVX delay objects that hold data of previous graph executions. Our existing tool flow converts the OpenVX graph in linear time into an Synchronous Dataflow (SDF) graph. SDF is a model of computation that abstracts from functionality and enables several prevalent analysis and scheduling techniques [15]. In this representation, nodes represent processing and the directed edges represent FIFO

buffers. The SDF nodes are annotated with the program size for each kernel in the OpenVX graph. If two kernels are linked by a data object in the OpenVX graph, the SDF nodes are connected by a directed edge annotated with the size of the data object. The resulting graph is a DAG. An example of this conversion is shown in Figure 1a and 1b.

In this work, we address a graph partitioning problem that arises when mapping the nodes of a DAG to the processing elements of a heterogeneous embedded multiprocessor. The processing elements (PEs) of this platform have a private local data memory and a separate program memory. A direct memory access controller is used to transfer data between the local memories and the external DDR memory of the system. The data memories have a size in the order of hundreds of kilobytes and can thus only store a small portion of the image. Therefore the input image is divided into *tiles*. The mode of operation of this hardware usually is that the nodes in the application graph are assigned to PEs and process the tiles one after the other. In most cases this can be pipelined such that while the PEs process the current tile, the direct memory access controller concurrently loads the next tile to the local memories and writes the processed tile from the previous iteration back to main memory.

However, this is only possible if the program memory size of the PEs is sufficient to store all kernel implementations. For the hardware platform under consideration it was found that this is not the case for more complex applications such as a Local Laplacian filter [18]. Therefore a gang scheduling [7] approach is used where the kernels are divided into groups of kernels (referred to as gangs) that do not violate memory constraints. Gangs are executed one after another on the target platform. After each execution, the kernels of the next gang are loaded. At no time any two kernels of different gangs are loaded in the program memories of the processors at the same time. Thus all intermediate data that is produced by the current gang but is needed by a kernel in a later gang needs to be transferred to external memory.

Since memory transfers, especially to external memories, are expensive in terms of power, the assignment of nodes to gangs is crucially important. There are many graph partitioning algorithms for the problem of dividing a graph into blocks of nodes under certain conditions [5]. However, in this case we require a strict ordering of gangs. Data objects may only be consumed in the same gang where they were produced and in gangs that are scheduled later. If this does not hold, there is no valid order in which the gangs can be executed on the platform. A valid order is a topological ordering of the graph that represents data dependencies between gangs. This graph can be created by taking the original DAG of the application and contracting all nodes that are assigned to the same gang into a single node. In order for a valid gang execution order to exist, the resulting graph therefore must be a DAG itself. An example for an incorrect assignment is shown in Figure 1c and a correct assignment in Figure 1d.

3 Preliminaries

In this section, we introduce the mathematical notation used throughout this paper, give the formal definition of the graph partitioning problem and show its relation to multiprocessor scheduling as a whole.

3.1 Basic Concepts

Let $G = (V = \{0, \dots, n-1\}, E, c, \omega)$ be an directed graph with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. We are looking for *blocks* of nodes $V_1, \dots, V_k$

that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. We call a block V_i *underloaded* [*overloaded*] if $c(V_i) < L_{\max}$ [if $c(V_i) > L_{\max}$]. If a node v has a neighbor in a block different of its own block then both nodes are called *boundary nodes*. An abstract view of the partitioned graph is the so-called *quotient graph*, in which nodes represent blocks and edges are induced by connectivity between blocks. The *weighted* version of the quotient graph has node weights which are set to the weight of the corresponding block and edge weights which are equal to the weight of the edges that run between the respective blocks.

3.2 Problem Definition

The partitions that we are looking for have to satisfy two constraints: a balancing constraint and a acyclicity constraint. The *balancing constraint* demands that $\forall i \in \{1..k\} : c(V_i) \leq L_{\max} := (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter $\epsilon \geq 0$. The *acyclicity constraint* mandates that the quotient graph is acyclic. The objective is to minimize the total *cut* $\sum_{i,j} w(E_{ij})$ where $E_{ij} := \{(u, v) \in E : u \in V_i, v \in V_j\}$. The *directed graph partitioning problem with acyclic quotient graph (DGPAQ)* is then defined as finding a partition $\Pi := \{V_1, \ldots, V_k\}$ that satisfies both constraints while minimizing the objective function.

3.3 Relation to Scheduling

The balancing constraint ensures that the size of the programs in a scheduling gang does not exceed the program memory size of the platform and thus is an important constraint for scheduling. Reducing the edge cut reduces the amount of data transfers between gangs and thus improves the memory bandwidth requirements of the application. Note that an application is either compute-limited (processors are always occupied) or bandwidth-limited (processors wait for data). Thus a minimization of transfers does not guarantee an optimal schedule. However, especially in embedded systems, the memory bandwidth is often the bottleneck and a schedule requiring a large amount of transfers will neither yield a good throughput nor good energy efficiency [17]. Therefore, we address the problem of minimizing the edge cut under the given constraints in isolation and do not solve a scheduling problem in this work. We provide linear-time heuristics that can later be employed as subroutines in broader scheduling algorithms to reduce data transfers.

3.4 Related Work

There has been a vast amount of research on graph partitioning so that we refer the reader to [23, 4, 5] for most of the material. Here, we focus on issues closely related to our main contributions. All general-purpose methods that are able to obtain good partitions for large real-world graphs are based on the multilevel principle. The basic idea can be traced back to multigrid solvers for systems of linear equations [24] but more recent practical methods are based on mostly graph theoretical aspects, in particular edge contraction and local search. There are many ways to create graph hierarchies such as matching-based schemes [27, 14, 19] or variations thereof [1] and techniques similar to algebraic multigrid, e.g. [16]. We refer the interested reader to the respective papers for more details. Well-known software packages based on this approach include Jostle [27], KaHIP [22], Metis [14] and Scotch [6]. However, none of these tools can partition directed graphs under the constraint that the quotient graph is a DAG. We are not aware of any related work that is able to satisfy this constraint.

Gang scheduling was originally introduced to efficiently schedule parallel programs with fine-grained interactions [7]. In recent work, this concept has been applied to schedule parallel applications on virtual machines in cloud computing [25] and extended to include hard

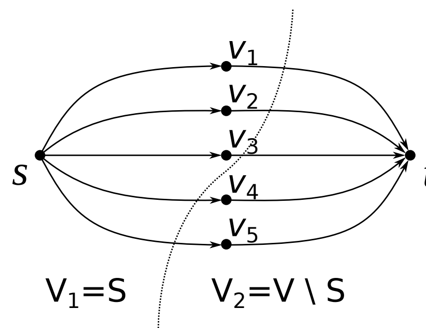


Figure 2 Reduction: subset sum problem is reduced to DGPAQ by creating a node for each a_i (the nodes in the center) and adding a source and sink node with edges as shown.

real-time tasks [10]. An important difference to our work is that in gang scheduling all tasks that exchange data with each other are assigned to the same gang, thus there is no communication between gangs. In our work, the limited program memory of embedded platforms does not allow to assign all kernels to the same gang. Therefore, there is communication between gangs which we aim to minimize by employing graph partitioning methods.

4 Hardness Results

In this section, we show that the problem under consideration is NP-complete when restricted to the case $k = 2$ and $\epsilon = 0$, and also hard to approximate with a finite approximation factor for $k \geq 3$. A given solution for an instance of DGPAQ can be verified in linear time by constructing the quotient graph $\mathcal{Q}$, checking the balance constraint and checking $\mathcal{Q}$ for acyclicity. The last task can be done in linear time in the size of $\mathcal{Q}$ using Kahn's algorithm [13]. We now reduce the subset sum problem to our problem. The proof is inspired by the reduction used in [20] which shows that the most balanced minimum cut problem is NP-complete.

▶ **Theorem 1.** *The DGPAQ problem is NP-complete for the bi-partitioning case with* $\epsilon = 0$.

Proof. We reduce the NP-complete [9] subset sum problem to DGPAQ. The decision version of the subset problem is stated as follows: Given a set of integers $\{a_1, \ldots, a_n\}$, is there a non-empty subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$ holds? The construction of an equivalent instance of DGPAQ is as follows: We construct a DAG $G = (V, E, c)$ with nodes $s, t \in V$ as well as a node $v_i \in V$ for each $i \in \{1, \ldots, n\}$. Then we set $A := \sum_i 2a_i$ and define the node weights as $c(s), c(t) := A$, $c(v_i) := 2a_i$. Afterwards, we insert edges $(s, v_i)\ \forall i$ and $(v_i, t)\ \forall i$. The graph is a DAG – an example topological ordering puts s first, t last and the remaining nodes at arbitrary positions in between. Figure 2 illustrates the construction. By definition, $L_{\max} = 3A/2$ for this instance of DGPAQ. Note that by construction A is divisible by 2. The construction can be done in polynomial time. Note that all balanced partitions $(S, V \backslash S)$ cut n edges, and due to the balance constraint s and t can never be in the same block. This ensures that there cannot be any edge (u, v) with $u \in V \backslash S$ and $v \in S$ and hence the quotient graph is acyclic. If the subset sum instance is a yes instance, then there is perfectly balanced bipartition and vice versa. ◀

The following theorem shows that it is not possible to find a finite factor approximation algorithm for our general problem where k is not a constant. The proof is a modification

of the proof by Andreev and Räcke [2] which shows this for the classical graph partitioning problem, i.e. no acyclicity constraint and for undirected inputs. Hence, we follow the proof of [2] closely with the difference being that the inputs that we construct are DAGs.

▶ **Theorem 2.** *The directed graph partitioning problem with acyclic quotient graph has no polynomial time approximation algorithm with a finite approximation factor for* $\epsilon = 0, k \geq 3$ *unless* $P = NP$.

Proof. The 3-Partition problem is defined as follows. Given $n = 3k$ integers $a_1, \dots, a_n$ and a threshold A such that $A/4 < a_i < A/2$ and $\sum_i a_i = kA$, decide whether the numbers can be partitioned into triples such that each triple adds up to A. This problem is *strongly* NP-complete [9], i.e. the problem remains NP-complete if all numbers a_i and A are polynomially bounded.

Now suppose we have an approximation algorithm for the directed graph partitioning problem with acyclic quotient graph for $\epsilon = 0$. We can use this algorithm to decide the 3-Partition problem with polynomially bounded numbers. To do so, we construct a graph G that contains n subgraphs. Subgraph i has a_i nodes. All weights are set to 1. We make each of the subgraphs a directed clique, i.e. all edges (u, v) with $u < v$ are inserted into the subgraph. By construction G is a DAG. This is the main difference to [2] in which the subgraphs are undirected cliques. Also since all numbers are polynomially bounded, the construction takes polynomial time.

Now, if the 3-Partition instance can be solved, the k-DGPAQ problem in G can be solved without cutting any edge. Note that this solution also fulfills the acyclicity constraint. If the 3-Partition instance cannot be solved, then the optimum solution to the k-DGPAQ problem will cut at least one edge. An approximation algorithm with finite approximation factor has to differentiate between these two cases. Hence, it can solve the 3-Partition problem. ◀

5 Heuristic Algorithms

In this section we present simple yet effective construction and local search heuristics to tackle the problem. Our general approach is as follows: First create an initial solution based on a topological ordering of the input graph and then apply a local search strategy to improve the objective of the solution while maintaining both constraints. We start the section with the construction algorithm and then present different local search heuristics.

5.1 Construction Algorithm

All of our local search heuristics start with an initial partitioning that fulfills both constraints, i.e. the quotient graph is acyclic and the balance constraint is satisfied. Our algorithm does this by computing a random topological ordering of the nodes using a modified version of Kahn's algorithm with randomized tie-breaking. More precisely, the algorithm initializes a list S with all nodes that have indegree zero and an empty list T. It then repeats the following steps until the list S is empty: Select a node from S uniformly at random and remove it from the list. Add the node to the tail of T. Remove all outgoing edges of the node. If this reduces the indegree of another node to zero, add it to S. When the algorithm terminates, the list T is a topological ordering of all nodes unless the graph has a cycle. Using list T, we can now derive initial solutions by dividing the graph into blocks of consecutive nodes w.r.t. the ordering. Due to the properties of the topological ordering there is no node in a block V_j that has an outgoing edge ending in a block V_i with $i < j$. Hence, the quotient graph of our solution is cycle-free. In addition, the blocks are chosen

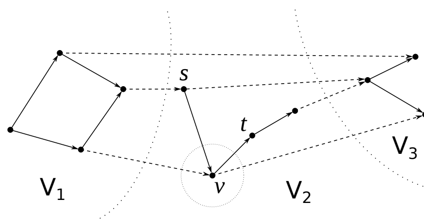


Figure 3 A DAG divided into three blocks. Internal edges are solid, external edges are dashed. Node v is a node that has non-zero internal and external cost for both C_{in} and C_{out}. Because of $(s, v) \in E \Rightarrow C_{in}(v, 2) > 0$, the node cannot be moved to V_1. Because of $(v, t) \in E \Rightarrow C_{out}(v, 2) > 0$, the node cannot be moved to V_3 either.

such that the balance constraint is fulfilled. There is obviously a large number of possible divisions. Our algorithm generates a balanced initial partitioning by dividing the ordering into blocks of size $\lfloor \frac{c(V)}{k} \rfloor$ or $\lceil \frac{c(V)}{k} \rceil$ uniformly at random. Since the construction algorithm is randomized, we run the heuristics ℓ times with different initial partitionings and pick the best solution afterwards.

5.2 Local Search Algorithms

Our local search heuristics take a given initial solution and move nodes between the blocks in order to decrease the edge cut. The reduction of the edge cut after a move is called the *gain* of the move. To compute the gain when moving node v, we define two functions:

$$C_{in}(v, i) := \omega(\{(u, v) \in E : u \in V_i\})$$
$$C_{out}(v, i) := \omega(\{(v, u) \in E : u \in V_i\})$$

Roughly speaking, C_{in} is the combined weight for all edges that start in nodes of block V_i and end in v. Analogously, C_{out} is the combined weight of all edges that start in v and connect to nodes in the block V_i. If $v \in V_i$, these costs are the weights of *internal* edges. These edges will become external edges and increase the objective if we move v to a different block. If $v \in V_j, j \neq i$, then these costs are weights of *external* edges, which will become internal and thus reduce the edge cut if v is moved to V_i. Figure 3 shows an example of internal and external edges.

We have multiple local search heuristics that differ in the size of the local search neighborhood: Simple Moves, Advanced Moves, Global Moves as well as FM moves. We found that the heuristics can often yield better results with a different initial partitioning. In order to compare the different heuristics, we will give each heuristic the same time budget and will restart the heuristics for different initial partitionings until it is exhausted.

5.2.1 Simple Moves (SM)

Simple moves start by picking a node v and moving it to a different block if this does not violate the constraints and improves the objective. Our simple move heuristic only considers to move a node $v \in V_i$ to adjacent blocks V_{i-1} and V_{i+1}. This is because there is a fast algorithm to check the *acyclicity constraint*. Assuming that the given solution is feasible with respect to both constraints, it is sufficient to check whether $C_{out}(v, i) = 0$ in the case that we want to move v to V_{i+1} and $C_{in}(v, i) = 0$ in the case that we want to move v to

V_{i-1}. The gain of a node movement depends on the block and is calculated as:

$$\begin{cases} C_{in}(v, i-1) - C_{out}(v, i) & \text{when moving } v \text{ to } V_{i-1} \\ C_{out}(v, i+1) - C_{in}(v, i) & \text{when moving } v \text{ to } V_{i+1}. \end{cases}$$

A block is eligible if the move does not create a cycle and does not overload the block. In addition, the gain has to be positive or zero but the balance of the partitioning is improved. If there is such a block, we move v to it. In the case that both blocks are eligible for the move and have the same gain, the heuristic selects one uniformly at random.

We repeat the process for all nodes. Our heuristic stops if there is no node with positive gain or balance cannot be improved. Hence, our heuristic terminates when a local minimum is found with respect to the local search neighborhood defined above. Note that even though the edge cut is not *strictly* monotonically decreasing, the combination of edge cut and difference in block weight is. In one pass, the heuristic considers the in- and outgoing edges of all nodes. Thus, each edge is considered exactly twice to calculate the gain for all nodes and the complexity of the heuristic is $O(m)$ per round.

5.2.2 Advanced Moves (AM)

This algorithm increases the local search neighborhood of the Simple Moves algorithm by considering more target blocks for a move. For the node $v \in V_i$ under consideration, all incoming edges are checked to find the node $u \in V_A$ where A is maximal. Also all outgoing edges are checked to find the node $w \in V_B$ where B is minimal. Since the original partition was obtained from a topological ordering, $A \le i \le B$ must hold, otherwise there would be back edges in the ordering and thus it would not be a topological ordering. If $A = i = B$, then the node v has in- and outgoing edges in its own block and cannot be moved. If $A < i$, then the node can be moved to blocks preceding V_i up to and including V_A in the topological ordering without creating a cycle. This is because all incoming edges of the node will either be internal to block V_A or are forward edges starting from blocks preceding V_A. Therefore it is still a topological ordering. However, when the node is moved to a block preceding V_A, the edge starting in this block becomes a back edge and the ordering is not a topological ordering anymore. Similar, if $i < B$, the node can be moved to blocks succeeding V_i up to and including V_B. Thus moving the node to V_j with $j \in \{A, \dots, B\} \setminus \{i\}$ will preserve the topological ordering of blocks. This is a sufficient condition to ensure the acyclicity constraint and is not computationally expensive to check. However, since it is not a necessary condition, it might prevent the heuristic from testing some possible moves. The Global Moves heuristic does not have this limitation, but has a higher computational complexity.

The gain of the moves to all allowed V_j is computed with the cost functions described in the previous section as $C_{in}(v, j) - C_{out}(v, i) + C_{out}(v, j) - C_{in}(v, i)$. In each iteration, the move with the largest gain such that the constraints are maintained is selected. Tie-breaking and gains of zero are handled in the same way as in Simple Moves.

This heuristic considers each edge exactly twice in order to calculate the gain when moving the node to any other block. Afterwards, a block yielding maximal gain is selected, which can be done in time proportional to the degree of a node. Thus, the complexity of this heuristic is $O(m)$.

5.2.3 Global Moves (GM)

With this algorithm, we increase the local search neighborhood even further by considering all other blocks. Starting from the initial partition, the algorithm computes the adjacency

lists of the quotient graph. Throughout the algorithm the quotient graph is kept up-to-date. When moving a node we update the adjacency information of the quotient graph and record whether a new edge has been created. If this is the case we check the quotient graph for acyclicity by using Kahn's algorithm and undo the last movement if it created a cycle.

The calculation of the gain values can be done in $O(m)$ as for the other heuristics. For a node, the heuristic needs to check the acyclicity constraint for all considered moves/blocks in the worst case. Since Kahn's algorithm checks the quotient graph for acyclicity, the total complexity of this heuristic is $O(m(m_{\mathcal{Q}}+k))$ where $m_{\mathcal{Q}}$ is the number of edges in the quotient graph. If the quotient graph is sparse, i.e. m_Q is $O(k)$, we get a complexity of $O(km)$.

5.2.4 FM Moves (FM)

This heuristic combines the quick check for acyclicity of the Advanced Moves heuristic with an adapted Fiduccia-Mattheyses algorithm [8] which gives the heuristic the ability to climb out of a local minimum. The initial partitioning is improved by exchanging nodes between a pair of blocks even if the gain is negative. The partition with the best objective that was seen during the pass will be returned. A pass starts with two blocks A and B, where A precedes B in the topological ordering of blocks. The algorithm will then calculate the gain for moving *enabled* boundary nodes to the other block. Using the same criterion to guarantee acyclicity as the Advanced Moves heuristic, we say that a boundary node is enabled if it is in A and does not have outgoing edges to nodes that precede B or it is in B and does not have incoming edges from nodes that follow A. The candidate moves, consisting of a gain and a node identifier, are inserted into a priority queue. The queue is a binary heap where the total order on the elements is implemented by comparing the gain of the moves and, if the gain is the same, a random number that is generated upon insertion.

In a loop that runs until the priority queue is depleted, the first move is extracted from the queue. If the selected move would overload the target block or is not enabled because it was disabled in a previous loop iteration, the heuristic continues with the next iteration. Otherwise, the move will be committed even if the gain is negative. The node is then locked, i.e. it cannot be moved again during this pass. This prevents thrashing and guarantees the termination of the algorithm. Unlike the Fiduccia-Mattheyses algorithm, a move in this scenario does not change the gain, it disables and enables other moves. For example, if a node w is moved from A to B, the heuristic will disable all nodes v in block B with $(w, v) \in E$ since they do not fulfill the condition for acyclicity anymore and moving any of them to A would introduce a back edge in the topological ordering of blocks. This does not necessarily mean that the quotient graph would become cyclic, however, assuring this would require a more expensive check like Kahn's algorithm. Note that the gain of the moves does not need to be re-calculated since w was locked and thus all nodes v will not be enabled again in this pass. On the other hand, moving w enables nodes in A if they are connected with an outgoing edge to w and if after the move they do not have other outgoing edges to blocks preceding B. The heuristic will calculate the gain for these nodes, enable and insert them into the priority queue. A move from B to A will enable and disable moves correspondingly. The loop will continue to move nodes between the blocks until the priority queue is depleted, which occurs when all nodes are either disabled or locked. Since the number of loop iterations is hard to predict due to the reinsertion of moves, it is limited to $2n/k$ which did not have a measurable impact on the quality of obtained partitionings. The best objective that was achieved in the pass is recorded. In the final step, the last moves are undone if required to reach the corresponding partitioning. This terminates the inner pass of the heuristic.

The outer pass of the heuristic will repeat the inner pass for randomly chosen pairs of blocks. At least one of these blocks has to be “active”. Initially, all blocks are marked as “active”. If and only if the inner pass results in movement of nodes, the two blocks will be marked as active for the next iteration. The heuristic stops if there are no more active blocks.

The overall time to compute gain values is $O(m)$. We now analyze the running time for a pair of blocks. In the worst case, all nodes of both blocks are enabled in the beginning and initializing the priority queue with $2n/k$ nodes requires $O(\frac{n}{k})$ time. Note that we cannot use a bucket priority queue, since the weights associated with the edges can be more or less arbitrarily distributed. Removing a node with the best gain from the queue takes $O(\log \frac{n}{k})$ time. If a move is committed in an iteration, the heuristic needs to calculate the gain of adjacent nodes. However, the heuristic will never calculate the gain of a move twice during a pass. Thus the total complexity of the inner pass is $O(\frac{n}{k} \log \frac{n}{k})$. Note that the inner pass needs to be performed for all pairs of blocks which yields overall time $O(m + m_{\mathcal{Q}} \frac{n}{k} \log \frac{n}{k})$ per round of the algorithm, or $O(m + n \log \frac{n}{k})$ if the quotient graph is sparse.

6 Experimental Evaluation

In this section we evaluate the performance of our algorithms. We start by presenting methodology and the systems we use for the evaluation. Then we evaluate the solution quality on small instances by comparing with the optimal solutions and evaluate our algorithms on complex imaging filters. We finish with testing the scalability of our algorithms.

Methodology. We have implemented the algorithms described above using C++. All programs have been compiled using g++ 4.8.0 and 32 bit index data types. The system we use is equipped with two Intel Xeon X5670 Hexa-Core processors (Westmere) running at a clock speed of 2.93 GHz. The machine has 128GB main memory, 12MB L3-Cache and 6×256KB L2-Cache. All instances described in this section will be made available on request.

Comparison with Optimal Solutions. This section compares the results of our heuristics against the optimal solution obtained by a non-polynomial time algorithm that performs an exhaustive search. We create a set of random graphs that are close to instances from typical applications. Our generation algorithm works by consecutively adding new graph levels with a random number of nodes. Each of the new nodes is connected to a random number of nodes in previous levels. Because the application domain of this work is imaging, we use a small number of input and output nodes (between one and three) which is typically the case for imaging and vision kernels (compare library of OpenVX vision functions [12]). Since the weight of nodes is representing the program size, we select a random value between the size of the smallest and the largest kernel in an implementation of the Local Laplacian filter for our target platform. The weight of edges is uniformly chosen between 1 and 100 to account for different sizes for intermediate buffers between the functions.

Because the following parameters have a major impact on the structure of the graph, we use two different values for each and generate 25 graphs for each of the eight resulting parameter combinations:

- The maximum size of a graph level is either set to a high value ($\sqrt{n}$) which results in a graph that can in extreme cases have $\sqrt{n}$ levels with about $\sqrt{n}$ nodes each, meaning that there is a high amount of data parallelism, and low values ($\sqrt[4]{n}$) such that the graph resembles more a long chain of nodes and thus represents the classical imaging pipeline with low data parallelism on kernel level.

Table 1 Each cell shows the averaged result of the heuristic for the current combination of block count k and imbalance ϵ. The value is the increase in cost compared to the optimal solution.

k	ϵ	SM	AM	GM	FM
2	20 %	3.41 %	3.41 %	3.41 %	0.26 %
	30 %	11.94 %	11.91 %	11.90 %	0.33 %
	40 %	14.71 %	14.78 %	14.58 %	1.29 %
	50 %	23.32 %	23.36 %	23.04 %	1.21 %
4	20 %	1.89 %	1.27 %	1.33 %	0.74 %
	30 %	4.03 %	3.22 %	3.25 %	0.67 %
	40 %	5.09 %	3.65 %	3.69 %	0.44 %
	50 %	6.50 %	4.04 %	4.19 %	0.31 %

Table 2 Table comparing the results of the manual implementation with the solution found by the heuristic.

	man.	SM	AM	GM	FM
number programs	20	22	16	16	17
number gangs	5	7	4	4	6
1-level edge cut	11,4	10,9	8,8	8,9	10,4
2-level edge cut	8,9	6,2	5,0	4,7	6,1
relative execution time	1,00	1,09	0,89	1,04	1,31

- The maximum number of edges is either set to the lowest number that ensures that inner nodes have at least one incoming and one outgoing edge and that the graph is connected or to $\sqrt{n}$ per node such that the number of edges scales with the problem size. This reflects applications with few and many data dependencies between functions.
- The maximum distance in terms of node indices, over which new nodes are connected to preceding nodes in the graph, is either set to a low value that results in a graph where nodes only have incoming edges from the closest preceding levels or it is set to n which means that there is no restriction on where edges can start. The first case models application where data is short-lived and only needed for the next step in a pipeline while the second case represents scenarios with a long data lifetime.

These 200 different problems instances were generated for problem sizes in the range of $n \in [10, \ldots, 20]$ nodes each. Table 1 shows the averaged approximation factor of the four heuristics when using a time budget of 10 milliseconds. The results show a good approximation of the optimal solution. The quality of SM, AM and GM degrades with large ϵ since they can get trapped in a local minimum, FM moves on the other hand shows a close and consistent approximation. The heuristics generally perform better on graphs that were created with more, unconstrained edges, presumably because there are more legal moves available of which the heuristic can pick the best one. We also found that the running time for a single pass of the heuristics is consistent across the instances while it varies drastically between milliseconds and several days for the exhaustive search. This emphasizes the need for a heuristic.

Local Laplacian Filter. The Local Laplacian filter is an edge-aware image processing filter. A detailed description of the algorithm and theoretical background is given in [18].

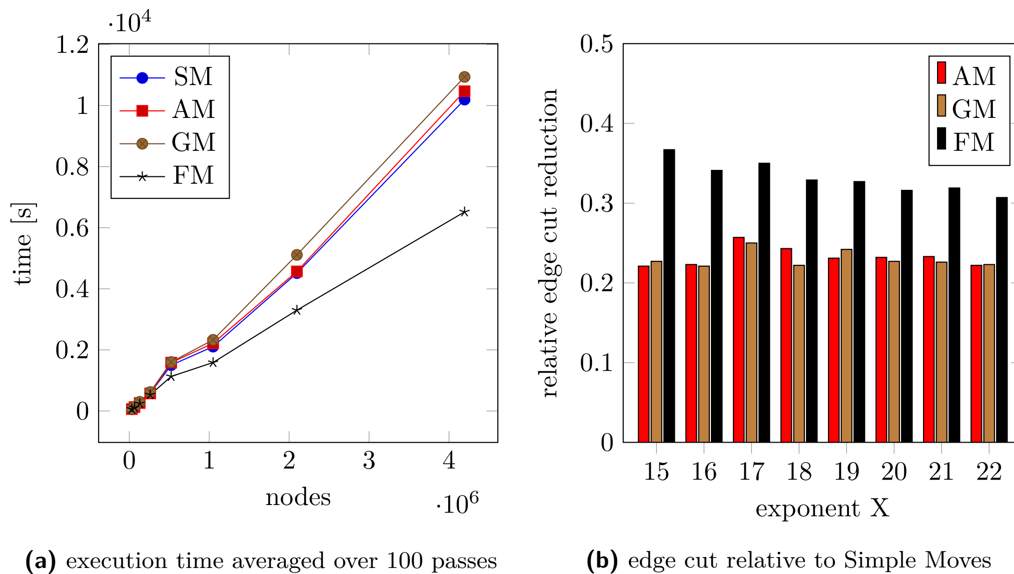
The algorithm uses concepts of *Gaussian pyramids* and *Laplacian pyramids* as well as a point-wise remapping function in order to enhance image details without creating arti-

facts. We model the data flow of the filter as a DAG where nodes represent simple function primitives, e.g. upsampling, downsampling and gaussian filtering for both image dimensions for each level of the pyramid generation. If there is a direct data dependency between two nodes, they are connected by an edge with a weight of the number of pixels of the corresponding buffer. The node weight is set to the program memory used by the primitive. The DAG has 72 nodes and 93 edges in total in our configuration. In an existing implementation, the primitives were grouped in a functional way (e.g. pyramid generation) by the developer into programs and then assigned to a total of five scheduling gangs. To evaluate the heuristics, we use a first pass with $L_{\max}$ set to the size of the program memory to find a good composition of function primitives into programs. The resulting quotient graph is then used in a second pass where $L_{\max}$ is set to the total number of PEs in order to find scheduling gangs that minimize external memory transfers. In this second step the acyclicity constraint is crucially important. In both passes, empty blocks were explicitly permitted to allow the heuristics to reduce the number of gangs. The time budget given to each heuristic is one minute. We also found that due to (desired) compiler optimizations, the final program memory size of a program can be smaller than the sum of its primitives. Since the entire process of partitioning, code generation and compilation is automated, we took advantage of this by slowly increasing ϵ until the programs became too large. The results are shown in Table 2. The 1-level edge cut shows the amount of communication between programs, the 2-level edge cut between gangs, both in megapixels.

If our heuristics are able to reduce bandwidth requirements of the application, the execution time will improve on all platforms where bandwidth is the limiting factor. Although this is the case for the majority of embedded platforms, we wanted to know what happens if we take bandwidth out of the equation. We obtained cycle counts for each program with a cycle-true compiled simulator of the hardware platform. Since context switching is synchronized, the longest execution time of a program in a gang equals the gang execution time assuming the programs never have to wait for data from external memories. The table shows the sum of this optimistic execution time for all gangs relative to the manual implementation.

All heuristics improve the edge cut by at least 30%, thus the schedule will be superior on all platforms where the manual implementation is bandwidth-limited. In addition, by reducing edge cut, the AM and GM heuristics find partitionings that require fewer gangs. For AM, this improves execution time, so the schedule will be superior even if bandwidth is not the limiting factor. For GM, the heuristic makes an additional choice that reduces edge cut even further, but does not balance compute-intensive programs well and thus execution time is not improved. In conclusion, we see a strong improvement in bandwidth demand and at the same time, with the exception of FM, an execution time that is on a par with or better than a manual implementation even if bandwidth is not a concern. We conclude that our pure edge cut reducing heuristics are a good starting point for the development of gang scheduling algorithms.

Random Geometric Graphs. We now look at the scalability of our heuristics. We do this on *random geometric graphs* where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost connected. These graphs were taken from [3] and were initially undirected. We convert them into DAGs by directing edges from smaller to larger node ids. The graph rgg$X$ has 2^X nodes. We vary $X \in [15, \ldots, 22]$. The allowed imbalance was set to 3% since this is one of the values used in [26]. Figure 4 shows



(a) execution time averaged over 100 passes

(b) edge cut relative to Simple Moves

Figure 4 Graph showing the execution time of each heuristic and the relative edge cut on directed random geometric graphs rggX.

the averaged time required for 100 passes of each heuristic and the relative improvement in edge cut that was found for $k = 8$ by the more advanced heuristics in comparison to the Simple Moves heuristic. The figure shows a linear growth in running time of our heuristics respective to number of nodes. The worst case complexity of FM moves was shown to be superlinear since it had to be assumed that all nodes are boundary nodes, which is not the case here. In fact, that FM only considers boundary nodes appears to improve the execution time compared to the other heuristics. We conclude that our algorithms scale well to large problems.

In another small experiment, we evaluated the quality of the solution found by the initial partitioning only. As expected, the best edge cut is always a fair amount larger than the one found by the heuristics, for example 29% compared to SM for the largest random geometric graph.

7 Conclusion

In this work we designed, implemented and evaluated new heuristics that partition streaming application graphs under constraints that are important for multiprocessor scheduling. It was shown that the constrained problem is NP-complete and that the heuristics yield good approximations of the optimal solution for small problem instances and have a linear growth for larger instances. In a simulation we could show the positive impact on communication volume of a real application for all heuristics and in one case even a reduction of execution time when bandwidth is not the limiting factor due to a reduction in number of scheduling gangs. The running time of the heuristics w.r.t. problem size is sufficient for this application domain, especially since the algorithms only need to run at compile-time. Also the communication volume was reduced by an extent that suggests that for future work it will be more rewarding to introduce additional objectives that help to improve gang execution time by better balancing compute-intensive kernels.

References

1. A. Abou-Rjeili and G. Karypis. Multilevel Algorithms for Partitioning Power-Law Graphs. In *Proc. of 20th IPDPS*, 2006.
2. K. Andreev and H. Räcke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.
3. D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, to appear.
4. C. Bichot and P. Siarry, editors. *Graph Partitioning.* Wiley, 2011.
5. A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering – Selected Topics,* to app.*, ArXiv:1311.3144*, 2014.
6. C. Chevalier and F. Pellegrini. PT-Scotch. *Parallel Computing*, 34(6-8):318–331, 2008. doi:10.1016/j.parco.2007.12.001.
7. D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and distributed Computing*, 16(4):306–318, 1992.
8. C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Conference on Design Automation*, pages 175–181, 1982.
9. M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.
10. J. Goossens and P. Richard. Optimal scheduling of periodic gang tasks. *Leibniz transactions on embedded systems*, 3(1):04–1, 2016.
11. Khronos Group. The OpenVX API. https://www.khronos.org/openvx/.
12. Khronos Group. The OpenVX Specification: Vision Functions. https://www.khronos.org/registry/OpenVX/specs/1.0/html/da/db6/group__group__vision__functions.html.
13. A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.
14. G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
15. E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
16. H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating Shape Optimizing Load Balancing for Parallel FEM Simulations by Algebraic Multigrid. In *Proc. of 20th IPDPS*, 2006.
17. P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, 2001.
18. S. Paris, S. W. Hasinoff, and J. Kautz. Local laplacian filters: edge-aware image processing with a laplacian pyramid. *ACM Trans. Graph.*, 30(4):68, 2011.
19. F. Pellegrini. Scotch Home Page. http://www.labri.fr/pelegrin/scotch.
20. J. C. Picard and M. Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Mathematical Programming Studies*, 13:8–16, 1980.
21. E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill. Addressing system-level optimization with openvx graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 644–649, 2014.
22. P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proc. of the 19th European Symp. on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

23 K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.

24 R. V. Southwell. Stress-Calculation in Frameworks by the Method of "Systematic Relaxation of Constraints". *Proc. of the Royal Society of London*, 151(872):56–95, 1935.

25 G. L. Stavrinides and H. D. Karatza. Scheduling different types of applications in a saas cloud. In *Proceedings of the 6th International Symposium on Business Modeling and Software Design (BMSD'16)*, pages 144–151, 2016.

26 C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

27 C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. American Mathematical Society, 2007.

Bilevel Programming Approaches to the Computation of Optimistic and Pessimistic Single-Leader-Multi-Follower Equilibria

Nicola Basilico[1], Stefano Coniglio[2], Nicola Gatti[3], and Alberto Marchesi[4]

1 **Dipartimento di Informatica, University of Milan, Milano, Italy**
nicola.basilico@unimi.it

2 **Department of Mathematical Sciences, University of Southampton, Southampton, UK**
s.coniglio@soton.ac.uk

3 **Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy**
nicola.gatti@polimi.it

4 **Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy**
alberto.marchesi@polimi.it

Abstract

We study the problem of computing an equilibrium in leader-follower games with a single leader and multiple followers where, after the leader's commitment to a mixed strategy, the followers play simultaneously in a noncooperative way, reaching a Nash equilibrium. We tackle the problem from a bilevel programming perspective. Since, given the leader's strategy, the followers' subgame may admit multiple Nash equilibria, we consider the cases where the followers play either the best (optimistic) or the worst (pessimistic) Nash equilibrium in terms of the leader's utility. For the optimistic case, we propose three formulations which cast the problem into a single level mixed-integer nonconvex program. For the pessimistic case, which, as we show, may admit a supremum but not a maximum, we develop an *ad hoc* branch-and-bound algorithm. Computational results are reported and illustrated.

1998 ACM Subject Classification G.1.6 [Optimization] Nonlinear programming

Keywords and phrases Stackelberg games, Nash equilibria, Game theory, Bilevel and nonlinear programming, Branch-and-bound

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.31

1 Introduction

In recent years, *leader-follower* (or *Stackelberg*) *games* have attracted a growing interest not just in game theory, but also in areas such as transportation science, security science, and combinatorial optimization. Such games model the interaction between rational agents (or players) in the context of sequential decision making. Considering, for simplicity, the two-player case, these games address situations where one agent plays first (the *leader*) and the other agent (the *follower*) plays second, after observing the *mixed strategy* the leader has committed to (a probability distribution over his/her actions). The algorithmic task is to compute an *equilibrium* (often called *solution* to the game), that is, a set of mixed strategies,

© Nicola Basilico, Stefano Coniglio, Nicola Gatti, and Alberto Marchesi;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 31; pp. 31:1–31:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

one per player, with the property that no player could obtain a larger utility by deviating from the equilibrium, provided that the other players act as the equilibrium prescribes.

In game theory, rich applications can be found in, among others, the security domain [2, 9]. A defender, whose aim is to protect a set of valuable targets from the attackers, plays first, as leader, while the attackers, acting as followers, observe the leader's defensive strategy and play second. Among different applications in combinatorial optimization, we mention interdiction problems [5, 13], toll setting problems [10], and network routing problems [1].

Most of the game-theoretical investigations on leader-follower games have, to the best of our knowledge, chiefly addressed the case of a single-follower. In that setting, it is known that the single follower can play, w.l.o.g., only a pure strategy (a probability distribution where a single action is played with probability 1), i.e., that there is always a pure strategy by which he/she can maximize his/her utility, and that computing an equilibrium is easy with complete information [17], while it becomes NP-hard for Bayesian games [7]. Algorithms are proposed in [7]. For what concerns games with more than two players, some works have investigated the case with multiple leaders and a single follower, see [11]. For the problem involving a single leader and multiple followers (the one on which we focus in this paper), only few results are available. It is known, for instance, that an equilibrium can be found in polynomial time if the followers play a correlated equilibrium [6], whereas the problem is NP-hard [7] if they play sequentially one at a time.

In this paper, we focus on the single-leader multi-follower case where the followers play *simultaneously* and *noncooperatively*, thus reaching a Nash Equilibrium (NE). We refer to an equilibrium in such games as to a *Single-Leader-Multi-Follower Nash Equilibrium* (SLMFNE). Computing such an equilibrium naturally amounts to solving a *bilevel programming* problem. Since, for a given mixed strategy of the leader, multiple NE might arise in the followers' subgame, we consider two cases: the *optimistic* one, where the followers play a NE which maximizes the leader's utility, and the *pessimistic* case, where the followers play a NE which minimizes it. Solving these two problems allows us to compute the tightest range of values the leader's utility can take independently of which NE is selected. We remark that, although the concept of SLMFNE has already been considered in the literature from a theoretical perspective, see, in particular, [17], no algorithmic methods to compute such an equilibrium are known.

The paper is organized as follows. The definition of the problem, its bilevel programming nature, and some of its properties are described in Section 2. We tackle the optimistic case in Section 3, where we construct three exact mixed-integer nonconvex mathematical programming formulations for it. The pessimistic case is addressed in Section 4, where we propose an *ad hoc* branch-and-bound method for its solution. Computational results are illustrated in Section 5, while Section 6 draws some concluding remarks.[1]

2 The problem

Consider a game with n players, with index set $N = \{1, \dots, n\}$. For each $p \in N$, let A_p be his/her set of actions, with $m_p := |A_p|$, and let the vector $x_p \in [0, 1]^{m_p}$, subject to $\sum_{a \in A_p} x_p^a = 1$, be the player's *strategy vector* (or strategy, for short). For each player $p \in N$, each component x_p^a of x_p corresponds to the probability by which action $a \in A_p$ is played. We call x_p a vector of *pure strategies* if $x_p \in \{0, 1\}^{m_p}$, or of *mixed strategies* if $x_p \in [0, 1]^{m_p}$. Throughout the paper, if not stated otherwise, we assume that all strategies are mixed. We

[1] An extended abstract of a preliminary version of this work appeared in [4].

also denote the collection of strategies of the different players, which forms a so-called *strategy profile*, by $x = (x_1, \dots, x_n)$.

We consider *normal form games* (a reference can be found in [16]). They are characterized by, for each $p \in N$, a multidimensional utility (or payoff) matrix $U_p \in \mathbb{Q}^{m_1 \times \dots \times m_n}$, whose components $U_p^{a_1,\dots,a_n}$ correspond to the utility obtained by player p when all the players play actions $a_1, \dots, a_n$. For a strategy profile $x = (x_1, \dots, x_n)$, the expected utility of player p is the multilinear function:

$$u_p(x_1, \dots, x_n) = \sum_{a_1 \in A_1} \dots \sum_{a_n \in A_n} U_p^{a_1,\dots,a_n} x_1^{a_1} \dots x_n^{a_n}.$$

With n players, this is an nth-degree polynomial.

According to the standard definition, a strategy profile $x = (x_1, \dots, x_n)$ is a Nash Equilibrium (NE) if, for each player $p \in N$ and for each strategy profile x' with $x'_q = x_q$ for all $q \in N \setminus \{p\}$ with, possibly, $x'_p \neq x_p$, the following inequality holds:

$$u_p(x_1, \dots, x_n) \geq u_p(x'_1, \dots, x'_n).$$

Intuitively, this is the same as imposing that, for each $p \in N$ and assuming the other players in $N \setminus \{p\}$ played as prescribed by the strategy profile x, player p would be unable to improve his/her utility when deviating from x_p by playing any other strategy $x'_p \neq x_p$.

In the remainder of the paper and for ease of notation, we consider the case of two followers (thus, with $n = 3$), assuming, w.l.o.g., $m_1 = m_2 = m_3 = m$. The nth player (player 3) takes the role of leader. The extension to the case of any $n > 3$ is, although notationally more involved, not difficult.

2.1 Problem definition

Computing a SLMFNE amounts to solving a so-called *bilevel programming problem with two followers*. Let, for each $p \in N$:

$$\Delta_p := \{x_p \in [0,1]^m : \sum_{a \in A_p} x_p^a = 1\}.$$

In the Optimistic case, we can compute a SLMFNE (O-SLMFNE) by solving:

$$\text{(O-SLMFNE)} \max_{\substack{(x_1,x_2,x_3) \in \\ \Delta_1 \times \Delta_2 \times \Delta_3}} \sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_3^{ijk} x_1^i x_2^j x_3^k \tag{1a}$$

$$\text{s.t.} \quad x_1 \in \operatorname*{argmax}_{x_1 \in \Delta_1} \left\{ \sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_1^{ijk} x_1^i x_2^j x_3^k \right\} \tag{1b}$$

$$x_2 \in \operatorname*{argmax}_{x_2 \in \Delta_2} \left\{ \sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_2^{ijk} x_1^i x_2^j x_3^k \right\}. \tag{1c}$$

Due to Constraints (1b)–(1c), the second level problems call for a pair (x_1, x_2) of followers' strategies forming a NE in the followers' subgame induced by the $x_3 \in \Delta_3$ chosen by the leader in the first level. Observe that, due to the definition of NE, the pair (x_1, x_2) is a NE for the given x_3 if and only if, at the same time, x_1 maximizes player 1's utility when assuming that player 2 would play x_2, and x_2 maximizes player 2's utility when assuming that player 1 would play x_1. Subject to those constraints, the first level calls for a triple (x_1, x_2, x_3) maximizing the leader's utility.

The problem is optimistic as, assuming that the second level admits many NE (x_1, x_2) for the chosen x_3, it calls for a pair (x_1, x_2) which, together with x_3, maximizes the leader's utility. Notice that, while any triple $(x_1, x_2, x_3) \in \Delta_1 \times \Delta_2 \times \Delta_3$ is a *feasible* solution to the problem as long as the pair (x_1, x_2) is a NE in the subgame induced by x_3, Problem (1a)–(1c) calls for a triple (x_1, x_2, x_3) which is *optimal*—as, if not, the leader would prefer to change his/her strategy and (x_1, x_2, x_3) would not be a SLMFNE.

In the Pessimistic case, computing a SLMFNE (P-SLMFNE) calls for a solution to the following problem:

$$\text{(P-SLMFNE)} \quad \max_{\substack{(x_1,x_2,x_3)\in \\ \Delta_1\times\Delta_2\times\Delta_3}} \min_{\substack{(x_1,x_2)\in \\ \Delta_1\times\Delta_2}} \sum_{i\in A_1} \sum_{j\in A_2} \sum_{k\in A_3} U_3^{ijk} x_1^i x_2^j x_3^k \tag{2a}$$

$$\text{s.t.} \quad \text{Constraints (1b), (1c).} \tag{2b}$$

This problem differs from its optimistic counterpart as, due to the assumption of pessimism, the leader here maximizes the *minimum* value taken by his/her utility over all pairs (x_1, x_2) which are NE in the followers' subgame induced by x_3.

2.2 Some properties of the problem

We observe that, as it is often the case in bilevel problems, the difference in leader's utility between an optimistic and a pessimistic SLMFNE can be arbitrarily large. Consider, for some $\lambda > 0$, a game with $n = 3$, $A_1 = \{i_1, i_2\}, A_2 = \{j_1, j_2\}$, $A_3 = \{k_1\}$, and utilities:

	j_1	j_2
i_1	1,1,λ	0,0,0
i_2	0,0,0	1,1,0

k_1

The unique O-SLMFNE in this game is (i_1, j_1, k_1), corresponding to a leader's utility of λ, while the unique P-SLMFNE is (i_2, j_2, k_1), with a leader's utility of 0. It follows that, for $\lambda \to \infty$, their difference in terms of leader's utility tends to ∞.

Differently from the O-SLMFNE case, where, as shown in [17], an equilibrium is always guaranteed to exist, this is not the case for P-SLMFNE—a behavior which can be observed in many pessimistic bilevel problems [18]. Consider a game with $n = 3$, $A_1 = \{i_1, i_2\}$, $A_2 = \{j_1, j_2\}$, $A_3 = \{k_1, k_2\}$. The matrices reported in the following are the utility matrices for, respectively, the case where the leader plays action k_1 with probability 1, action k_2 with probability 1, or the strategy vector $x_3 = (1 - \rho, \rho)$ for some $\rho \in (0, 1)$ (the latter matrix is the convex combination of the first two with weights x_3):

	j_1	j_2
i_1	1,1,0	2,2,5
i_2	$\frac{1}{2}$,$\frac{1}{2}$,1	1,1,0

k_1

	j_1	j_2
i_1	0,0,0	2,2,10
i_2	$\frac{1}{2}$,$\frac{1}{2}$,1	0,0,0

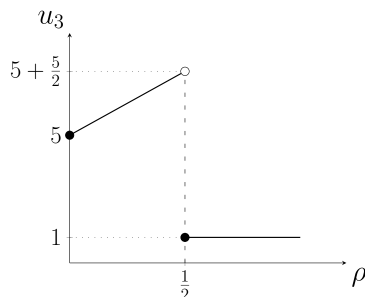
k_2

	j_1	j_2
i_1	$1-\rho$,$1-\rho$,0	2,2,$5+5\rho$
i_2	$\frac{1}{2}$,$\frac{1}{2}$,1	$1-\rho$,$1-\rho$,0

$x_3 = (1 - \rho, \rho)$

In the optimistic case, (i_1, j_2, k_2) is the unique SLMFNE (it is the only pure one and, as it achieves the leader's largest utility in U_3, mixed strategies cannot yield a better utility).

In the pessimistic case, if the leader played k_2, the followers would respond, among the two NE (i_2, j_1) and (i_1, j_2), with (i_2, j_1) (so to minimize the leader's utility). If the leader played $x_3 = (1 - \rho, \rho)$, we would obtain the subgame in the third matrix. For $\rho < \frac{1}{2}$, (i_1, j_2) is the unique NE, giving the leader a utility of $5 + 5\rho$. For $\rho \geq \frac{1}{2}$, we have again the two NE (i_1, j_2) and (i_2, j_1), with a utility of, respectively, $5 + 5\rho$ and 1. Since the latter is selected in the pessimistic case, we conclude that the game admits no pessimistic SLMFNE. This is because the leader's utility (and the optimization problem), when written as a function of ρ, achieves a supremum at $\rho = \frac{1}{2}$, but not a maximum. See the following graph for an illustration.



From a combinatorial perspective, the hardness and inapproximability (in polynomial time, up to within a constant factor) of the problem of computing a SLMFNE is a direct consequence of the NP-hardness and inapproximability of computing, in a two-player game, a NE which maximizes the sum of the players' utilities [8]. Indeed, the latter problem is directly reduced to the computation of a SLMFNE for the case where the leader has a single action and his/her utility is equal to plus (in the optimistic case) or minus (in the pessimistic case) the sum of the utilities of the followers.

3 Optimistic case

We propose three Mixed-Integer NonLinear Programming (MINLP) formulations to compute a SLMFNE in the optimistic case.

3.1 MINLP-I

To obtain a first single level formulation for the problem, we proceed as follows, applying a standard reformulation [16] involving complementarity constraints.

Let, for all $i \in A_1$ and $j \in A_2$, $\tilde{U}_1^{ij} := \sum_{k \in A_3} U_1^{ijk} x_3^k$ and $\tilde{U}_2^{ij} = \sum_{k \in A_3} U_2^{ijk} x_3^k$ be the matrices of the followers' subgame, parameterized by x_3. For (x_1, x_2) to be a NE, x_1 must be an optimal solution to the Linear Program (LP):

$$\max_{x_1 \in \Delta_1} \left\{ \sum_{i \in A_1} \sum_{j \in A_2} \tilde{U}_1^{ij} x_1^i x_2^j \right\},$$

where $\tilde{U}_1^{ij} x_1^i x_2^j$ is a linear function of x_1. Since the LP is feasible and bounded for any $x_2 \in \Delta_2$, we have, by complementary slackness, that $x_1 \in \Delta_1$ is optimal if and only if there is a scalar v_1 such that, for all $i \in A_1$:

$$\left(v_1 - \sum_{j \in A_2} \tilde{U}_1^{ij} x_2^j\right) x_1^i = 0$$

$$v_1 \geq \sum_{j \in A_2} \tilde{U}_1^{ij} x_2^j.$$

Applying a similar reasoning to x_2, we obtain that $x_2 \in \Delta_2$ is optimal if and only if there is a scalar v_2 such that, for all $j \in A_2$:

$$\big(v_2 - \sum_{i \in A_1} \tilde{U}_2^{ij} x_1^i\big) x_2^j = 0$$

$$v_2 \geq \sum_{i \in A_1} \tilde{U}_2^{ij} x_1^i.$$

We conclude that (x_1, x_2) is a NE if and only if there are $v_1, v_2 \geq 0$ such that x_1 and x_2 simultaneously satisfy these four conditions.

After substituting for $\tilde{U}_1$ and $\tilde{U}_2$ their linear expressions in x_3, we obtain, for player 1 and for all $i \in A_1$, constraints:

$$\left(v_1 - \sum_{j \in A_2} \sum_{k \in A_3} U_1^{ijk} x_2^j x_3^k\right) x_1^i = 0$$

$$v_1 \geq \sum_{j \in A_2} \sum_{k \in A_3} U_1^{ijk} x_2^j x_3^k$$

and, for player 2 and for all $j \in A_2$, constraints:

$$\left(v_2 - \sum_{i \in A_1} \sum_{k \in A_3} U_2^{ijk} x_1^i x_3^k\right) x_2^j = 0$$

$$v_2 \geq \sum_{i \in A_1} \sum_{k \in A_3} U_2^{ijk} x_1^i x_3^k.$$

By imposing them *in lieu* of the two second level argmax constraints of Problem (1), that is, Constraints (1b)–(1c), we obtain a continuous single level formulation with nonconvex trilinear terms.[2][3]

3.2 MINLP-II

What we propose now is aimed at achieving a formulation which can be solved more efficiently.

Since each term of the complementarity constraints we introduced is bounded, we can apply a simple reformulation. Letting $s_1 \in \{0,1\}^m$ and $s_2 \in \{0,1\}^m$ be the *antisupport vectors* of x_1 and x_2, it suffices to impose, for all $i \in A_1$:

$$x_1^i \leq 1 - s_1^i$$

$$v_1 - \sum_{j \in A_2} \sum_{k \in A_3} U_1^{ijk} x_2^j x_3^k \leq M s_1^i$$

and, for all $j \in A_2$:

$$x_2^j \leq 1 - s_2^j$$

$$v_2 - \sum_{i \in A_1} \sum_{k \in A_3} U_2^{ijk} x_1^i x_3^k \leq M s_2^j,$$

where M is an upper bound on the entries of U_1, U_2. This way, while still retaining the original trilinear objective function, only bilinear constraints are needed.

[2] Note that strong duality can be employed in place of complementary slackness. Preliminary experiments, though, suggest that the second option is computationally preferable.

[3] We remark that, in this form, the problem correspond to a Mathematical Program with Equilibrium Constraints (MPEC). The interested reader can find more references to this type of problems in [12].

3.3 MINLP-III

Ultimately, we aim to solve the problem with spatial branch-and-bound techniques, such as those implemented in SCIP. The main strategy of such methods to handle nonlinearities is to isolate "simple" nonlinear terms (bilinear or trilinear in our case) by shifting them into a new (so-called *defining*) constraint to which convex envelopes are applied.

We propose to anticipate this reformulation, so to be able to derive some valid constraints. First, we introduce:
(i) variable y_{23}^{jk} and constraint $y_{23}^{jk} = x_2^j x_3^k$ for all $j \in A_2, k \in A_3$,
(ii) variable y_{13}^{ik} and constraint $y_{13}^{ik} = x_1^i x_3^k$ for all $i \in A_1, k \in A_3$,
(iii) variable z^{ijk} and constraint $z^{ijk} = x_1^i y_{23}^{jk}$ for all $i \in A_1, j \in A_2, k \in A_3$.
Then, by substituting each bilinear and trilinear term with the newly introduced variables, we obtain a formulation which is linear everywhere, except for the defining constraints.

We now observe that, by definition, the matrix $\{y_{23}^{jk}\}_{jk \in A_2 \times A_3}$ is the outer product of the stochastic vectors x_2 and x_3 and, as such, is a stochastic matrix itself. The same holds for the tensor $\{z^{ijk}\}_{ijk \in A_1 \times A_2 \times A_3}$, which is the outer product of the vectors x_1, x_2, x_3 and, as such, is a stochastic tensor. This implies the validity of the following three constraints:

$$\sum_{i \in A_1} \sum_{k \in A_3} y_{13}^{ik} = 1$$
$$\sum_{j \in A_2} \sum_{k \in A_3} y_{23}^{jk} = 1$$
$$\sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} z^{ijk} = 1.$$

The final single level mixed-integer nonconvex formulation that we propose for O-SLMFNE thus reads:

$$\begin{aligned}
\max \ & \sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_3^{ijk} z^{ijk} \\
\text{s.t. } & v_1 - \sum_{j \in A_2} \sum_{k \in A_3} U_1^{ijk} y_{23}^{jk} \leq M s_1^i \quad \forall i \in A_1 \\
& v_2 - \sum_{i \in A_1} \sum_{k \in A_3} U_2^{ijk} y_{13}^{ik} \leq M s_2^j \quad \forall j \in A_2 \\
& x_1^i \leq 1 - s_1^i \quad \forall i \in A_1 \\
& x_2^j \leq 1 - s_2^j \quad \forall j \in A_2 \\
& v_1 \geq \sum_{j \in A_2} \sum_{k \in A_3} U_1^{ijk} y_{23}^{jk} \quad \forall i \in A_1 \\
& v_2 \geq \sum_{i \in A_1} \sum_{k \in A_3} U_2^{ijk} y_{13}^{ik} \quad \forall j \in A_2 \\
& y_{13}^{ik} = x_1^i x_3^k \quad \forall (i,k) \in A_1 \times A_3 \\
& y_{23}^{jk} = x_2^j x_3^k \quad \forall (j,k) \in A_2 \times A_3 \\
& z^{ijk} = x_1^i y_{23}^{jk} \quad \forall (i,j,k) \in A_1 \times A_2 \times A_3 \\
& \sum_{i \in A_1} \sum_{k \in A_3} y_{13}^{ik} = 1 \\
& \sum_{j \in A_2} \sum_{k \in A_3} y_{23}^{jk} = 1 \\
& \sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} z^{ijk} = 1 \\
& (x_1, x_2, x_3) \in \Delta_1 \times \Delta_2 \times \Delta_3 \\
& y_{13}, y_{23} \in [0,1]^{m \times m} \\
& z \in [0,1]^{m \times m \times m} \\
& s_1, s_2 \in \{0,1\}^m \\
& v_1, v_2 \text{ free.}
\end{aligned}$$

4 Pessimistic case

For P-SLMFNE, we introduce an *ad hoc* method based on branch-and-bound. For simplicity, we first describe it for the case where the followers are restricted to pure strategies, with the extension to the mixed case following next.

4.1 Basic enumerative idea and outcome configurations

The key ingredient of the method is what we call *outcome configuration.* Given a strategy vector $x_3 \in \Delta_3$, we call a pair (S^+, S^-), with $S^+ \subseteq A_1 \times A_2$ and $S^- = A_1 \times A_2 \setminus S^+$, *outcome configuration* if, for the given x_3, it satisfies two constraints:

(i) all pairs of actions in S^+ correspond to a NE in the followers' subgame;
(ii) all pairs in S^- do not.

Given a pair (S^+, S^-), a strategy vector $x_3 \in \Delta_3$ such that i) and ii) are satisfied guarantees that, if the leader played x_3, the set of NE in the followers' subgame would coincide with S^+. Since, given an outcome configuration (S^+, S^-), the leader's utility at each NE in S^+ is a (linear) function of x_3, we must then look (due to the pessimistic setting) for an x_3 which iii) maximizes the smallest value taken by the leader's utility over S^+.

By constructing (we will propose a more efficient method later on) all the pairs (S^+, S^-) in $2^{A_1 \times A_2}$ and computing (if it exists) the corresponding x_3, a P-SLMFNE (if it exists) is obtained by choosing the triple (S^+, S^-, x_3) which gives the leader the largest utility (and, then, selecting the NE in S^+ which minimizes the leader's utility).

Let us discuss how to compute x_3 for a given (S^+, S^-). By definition of NE, constraints i) can be expressed as the following set of inequalities, which are linear in x_3:

$$\sum_{k \in A_3} U_1^{ijk} x_3^k \geq \sum_{k \in A_3} U_1^{i'jk} x_3^k \quad \forall (i,j) \in S^+, i' \in A_1 \setminus \{i\} \tag{3a}$$

$$\sum_{k \in A_3} U_2^{ijk} x_3^k \geq \sum_{k \in A_3} U_2^{ij'k} x_3^k \quad \forall (i,j) \in S^+, j' \in A_2 \setminus \{j\}. \tag{3b}$$

Given some sufficiently small $\epsilon > 0$, Constraints ii) can be (approximately) written as the following disjunction:

$$\bigvee_{i' \in A_1 \setminus \{i\}} \left(\sum_{k \in A_3} U_1^{ijk} x_3^k + \epsilon \leq \sum_{k \in A_3} U_1^{i'jk} x_3^k \right) \quad \bigvee$$
$$\bigvee_{j' \in A_2 \setminus \{j\}} \left(\sum_{k \in A_3} U_2^{ijk} x_3^k + \epsilon \leq \sum_{k \in A_3} U_2^{ij'k} x_3^k \right) \qquad \forall (i,j) \in S^-. \tag{4}$$

For every $(i,j) \in S^-$, the disjunction imposes the existence of an action i' of player 1 giving him/her a utility larger than that obtained when playing i by, at least, some $\epsilon > 0$, assuming the other player played j (or *vice versa* for player 2 and an action j').

Note that each term of the disjunction should be strict, as the disjunction represents the complement of a polytope. The approximation with ϵ has the role of preventing x_3 from reaching one of the breakpoints of the leader's utility function (see the illustration in Section 2), where the pessimistic problem always achieves a supremum but not a maximum.

We can cast Constraints (4) in terms of Mixed-Integer Linear Programming (MILP) by introducing a binary variable per term of the disjunction, with a constraint requiring its sum to be 1.[4] After introducing the binary variables $y_{iji'} \in \{0,1\}$, for $i' \in A_1 \setminus \{i\}$, and

[4] Due to considering polytopes, the extended LP formulation of Balas [3] could be used. Nevertheless, we have found the MILP approach computationally affordable.

$y_{ijj'} \in \{0,1\}$, for $j' \in A_2 \setminus \{j\}$, the reformulation reads:

$$\sum_{k \in A_3} U_1^{ijk} x_3^k + \epsilon \leq \sum_{k \in A_3} U_1^{i'jk} x_3^k + M(1 - y_{iji'}) \qquad \forall i' \in A_1 \setminus \{i\} \tag{5a}$$

$$\sum_{k \in A_3} U_1^{ijk} x_3^k + \epsilon \leq \sum_{k \in A_3} U_2^{ij'k} x_3^k + M(1 - y_{ijj'}) \qquad \forall j' \in A_2 \setminus \{j\} \tag{5b}$$

$$\sum_{i' \in A_1 \setminus \{i\}} y_{iji'} + \sum_{j' \in A_2 \setminus \{j\}} y_{ijj'} = 1, \tag{5c}$$

where M is a (previously introduced) upper bound on the entries of U_1, U_2.

A solution satisfying i) and ii) which is also optimal in the sense of iii) is then found by solving the following MILP subproblem:

$$\max_{\substack{\eta \text{ free} \\ x_3 \in \Delta_3}} \left\{ \begin{array}{ll} \eta: \quad \eta \leq \sum_{k \in A_3} U_3^{ijk} x_3^k & \forall (i,j) \in S^+ \\ \qquad \text{Constraints (3), (4).} & \end{array} \right\} \tag{6}$$

4.2 Branch-and-Bound approach

We now propose an alternative method which does not require to carry out the complete enumeration of the elements of $2^{A_1 \times A_2}$.

Given a strategy vector $x_3 \in \Delta_3$, we call a pair (S^+, S^-) *relaxed outcome configuration* if $S^- \subseteq (A_1 \times A_2) \setminus S^+$, differently from the previous definition where $S^- = (A_1 \times A_2) \setminus S^+$.

Notice that, when $S^- \subset (A_1 \times A_2) \setminus S^+$, it is not always the case that, when the leader plays a solution x_3 to Subproblem (6), the only NE in the followers' subgame are those in S^+. Indeed, due to $S^+ \cup S^- \subset A_1 \times A_2$, the followers' subgame may admit another NE $(i', j') \in A_1 \times A_2 \setminus S^+ \setminus S^-$ which provides the leader with a strictly smaller utility than that he/she would receive from the pairs of NE in S^+. Since, whenever this is the case, (i', j') would be part of the P-SLMFNE corresponding to x_3, the correctness of the method would be lost.

To verify whether this is the case, i.e., whether one such (i', j') exists, it suffices to carry out an operation which we refer to as *feasibility check*. We solve the followers' subgame (in the pessimistic sense) for the given x_3, and compare the NE (i', j') thus found to the worst one in S^+ (when working with pure strategies only, this check can be done in $O(m^2)$). If $(i', j') \notin S^+$, we resort to branching. More precisely, to account for the case where (i', j') is a NE, we introduce a *left node* (S_L^+, S_L^-) with $S_L^+ := S^+ \cup \{(i', j')\}$ and $S_L^- := S^-$, whereas, to account for the case where (i', j') is not a NE, we introduce a *right node* (S_R^+, S_R^-) with $S_R^+ := S^+$ and $S_R^- := S^- \cup \{(i', j')\}$.

We remark that, as a consequence of $S^- \subseteq (A_1 \times A_2) \setminus S^+$, optimal solutions to Subproblem (6) always yield an upper bound on the utility the leader would obtain with a strategy x_3 by which all pairs of followers' actions in S^+ constitute a NE, while all pairs in S^- do not.

At the root node, we solve the optimistic problem, obtaining a triple $(x_1 = e_i, x_2 = e_j, x_3)$. If, by feasibility check, we find a pair $(i', j') = (i, j)$ (or a different one, but yielding the same utility), the problem is solved. If not, we create two nodes: (S_L^+, S_L^-) with $S_L^+ = \{(i', j')\}, S_L^- = \emptyset$, and (S_R^+, S_R^-) with $S_R^+ = \emptyset, S_R^- = \{(i', j')\}$. Since Subproblem (6) is not well–defined for (S_R^+, S_R^-) as, in it, $S_R^+ = \emptyset$, whenever $S^+ = \emptyset$ in a (S^+, S^-) pair, we solve, *in lieu* of Subproblem (6), one of the formulations we proposed for O-SLMFNE with the further addition of Constraints (4) for all pairs in S^-. This way, we can find an upper bound also for the case where S^+ is empty.

4.3 Extension to the unrestricted case

To extend the method to the unrestricted mixed case, assume now that the components of each pair (S^+, S^-) are, rather than pairs of actions, disjoint sets of pairs of strategy vectors $(\bar{x}_1, \bar{x}_2)$ of the followers, with $(\bar{x}_1, \bar{x}_2) \in \Delta_1 \times \Delta_2$.

Constraints i) should now impose all strategy vectors in $(\bar{x}_1, \bar{x}_2) \in S^+$ to be NE. Since the utility obtained with any strategy vector x_p is a convex combination with weights x_p of those obtained with pure strategies, it suffices to impose, for each follower, that $(\bar{x}_1, \bar{x}_2)$ should yield a utility at least as large as that obtained with any pure strategy. We arrive at the following linear (in x_3) inequalities:

$$\sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_1^{ijk} \bar{x}_1^i \bar{x}_2^j x_3^k \geq \sum_{j \in A_2} \sum_{k \in A_3} U_1^{i'jk} \bar{x}_2^j x_3^k \quad \forall (\bar{x}_1, \bar{x}_2) \in S^+, i' \in A_1 \tag{7a}$$

$$\sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_2^{ijk} \bar{x}_1^i \bar{x}_2^j x_3^k \geq \sum_{i \in A_1} \sum_{k \in A_3} U_2^{ij'k} \bar{x}_1^i x_3^k \quad \forall (\bar{x}_1, \bar{x}_2) \in S^+, j' \in A_2. \tag{7b}$$

We can apply a similar argument when stating constraints ii) which, in the mixed case, impose that all strategy vectors in S^- are not NE. Indeed, it suffices to require the existence of an action $i' \in A_1$ or of an action $j' \in A_2$ providing the respective player with a strict improvement to his/her utility. Let $\pi(x_p) := \{a\}$ if $x_p^a = 1$, $\emptyset$ otherwise. For a given, sufficiently small, $\epsilon > 0$, the constraints can be (approximately) written as the following disjunction:

$$\bigvee_{i' \in A_1 \setminus \pi(\bar{x}_1)} \left(\sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_1^{ijk} \bar{x}_1^i \bar{x}_2^j x_3^k + \epsilon \leq \sum_{j \in A_2} \sum_{k \in A_3} U_1^{i'jk} \bar{x}_2^j x_3^k \right) \quad \bigvee$$
$$\bigvee_{j' \in A_2 \setminus \pi(\bar{x}_2)} \left(\sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_2^{ijk} \bar{x}_1^i \bar{x}_2^j x_3^k + \epsilon \leq \sum_{i \in A_1} \sum_{k \in A_3} U_2^{ij'k} \bar{x}_1^i x_3^k \right) \forall (\bar{x}_1, \bar{x}_2) \in S^-, \tag{8}$$

which can be rewritten as a MILP as previously discussed for the restricted case.

A strategy vector x_3 satisfying the previous constraints which is also optimal in the sense of iii) is found by solving the following MILP:

$$\max_{\substack{\eta \text{ free} \\ x_3 \in \Delta_3}} \left\{ \begin{array}{ll} \eta : & \eta \leq \sum_{i \in A_1} \sum_{j \in A_2} \sum_{k \in A_3} U_3^{ijk} \bar{x}_1^i \bar{x}_2^j x_3^k \qquad \forall (\bar{x}_1, \bar{x}_2) \in S^+ \\ & \text{Constraints (7), (8)} \end{array} \right\}. \tag{9}$$

The initialization of the search tree and the solution of nodes with $S^+ = \emptyset$ can be carried out as for the restricted case. Differently from that case though, we cannot perform feasibility check in $O(m^2)$ by inspection. For it, we resort to solving one of the formulations we gave for O-SLMFNE with a given, fixed x_3, after changing the sign of the objective function into a minus (so to consider the pessimistic case).

We observe that, differently from the restricted case, the search tree might not be finite when mixed strategies are considered. This is due to fact that a game may contain uncountably many triples (x_1, x_2, x_3) where (x_1, x_2) is a NE in the followers' subgame. As a consequence, the branch-and-bound algorithm might not terminate. Note, though, that, by halting the method after any finite amount of iteration of computing time, one would nevertheless be able to obtain a lower and an upper bound to the problem.

Table 1 Results obtained when computing an O-SLMFNE with the three proposed MINLP formulations.

	MINLP–I					**MINLP–II**					**MINLP–III**				
m	Time	Gap	UB	LB	Sol	Time	Gap	UB	LB	Sol	Time	Gap	UB	LB	Sol
4	3600	46.5	100	53.5	80	3600	11.0	100	89.0	100	1	0.0	91.4	91.4	100
6	3600	93.2	100	6.8	10	3600	53.0	100	47.0	80	413	10.7	92.8	82.9	90
8	3600	91.0	100	9.0	10	3600	53.8	100	46.2	70	987	0.7	96.8	96.1	100
10	3600	99.0	100	1.0	0	3600	55.0	100	45.0	70	2147	22.1	99.0	77.1	80
12	3600	99.0	100	1.0	0	3600	64.7	100	35.3	60	3242	7.7	99.7	92.0	100
14	3600	99.0	100	1.0	0	3600	62.3	100	37.7	50	3240	11.8	99.8	88.1	100
16	3600	99.0	100	1.0	0	3600	92.3	100	7.7	10	3243	18.3	99.9	81.6	100
18	3600	99.0	100	1.0	0	3600	93.1	100	6.9	10	2887	7.1	99.9	92.8	100
20	3600	99.0	100	1.0	0	3600	79.2	100	20.8	30	3265	15.2	99.9	84.8	100
25	3600	99.0	100	1.0	0	3600	84.3	100	15.7	40	3600	15.7	100.0	84.3	100
30	3600	99.0	100	1.0	0	3600	99.0	100	1.0	0	3600	14.5	100.0	85.5	100
35	3600	99.0	100	1.0	0	3600	99.0	100	1.0	0	3600	16.7	100.0	83.3	100
40	3600	99.0	100	1.0	0	3600	99.0	100	1.0	0	3400	14.4	100.0	85.6	100
45	3600	99.0	100	1.0	0	3600	99.0	100	1.0	0	3600	22.9	100.0	77.1	90
Avg	3600	94.3	100	5.7	7	3600	74.6	100	25.4	37	2659	12.7	98.5	85.9	97

5 Computational results

We consider a testbed of instances constructed with GAMUT, a widely adopted suite of game generators [14], of class `Uniform Random Games`. We assume the same number of actions m for each agent, with $U_1, U_2, U_3 \in [1, 100]^{m\times m\times m}$, and construct 10 different instances per value of m. We experiment on games with $n = 3$ players, with $m = 4, 6, 8, 10, 12, 14, 16, 18, 20, 25, 30, 35, 40, 45$ actions. In the upcoming Tables 1 and 2, we report, for each value of m and on average over the 10 corresponding game instances, the following four values:

1. computing time (Time), also including instances for which the time limit is reached,
2. optimality gap (Gap, defined as $\frac{UB-LB}{UB}100\%$),
3. upper bound (UB),
4. lower bound (LB),

plus a fifth value which, rather than an average, reports, for a given value of m, the:

5. percentage of games for which a feasible solution is found (Sol).

The experiments are run on a UNIX machine with a total of 32 cores working at 2.3 GHz, equipped with 128 GB of RAM, within a time limit of 3600 seconds per game, on a single thread. We assume, throughout the section, that leader and followers are entitled to mixed strategies, both in the optimistic and pessimistic cases.

5.1 Optimistic case

We experiment with the three MINLP formulations we proposed in Section 3, solving them with the spatial branch-and-bound solver SCIP 3.2.1. The results are reported in Table 1.

The results confirm that reformulating the complementarity constraints via binary variables yields (as expected) a considerable improvement, reducing the gap from 94.3% (MINLP-I) to 74.6% (MINLP-II), on average. The reformulation with additional constraints carried out to obtain MINLP-III allows for a very substantial improvement, bringing the gap down to 12.7%, on average. Although the UB is not substantially improved by MINLP-III, reaching

Table 2 Results obtained when computing a P-SLMFNE with the proposed branch-and-bound algorithm, for $\epsilon = 0.1, 1, 5$.

	$\epsilon = 0.1$					$\epsilon = 1$					$\epsilon = 5$				
m	Time	Gap	UB	LB	Sol	Time	Gap	UB	LB	Sol	Time	Gap	UB	LB	Sol
4	2525	25.9	88.5	65.6	100	362	0.0	83.0	83.0	100	37	5.6	84.9	80.2	100
6	3600	41.6	89.9	52.5	100	3281	34.2	88.2	58.0	90	2247	11.5	85.7	75.8	100
8	3600	49.3	96.2	48.8	100	3600	42.7	95.4	54.6	100	3600	28.8	90.1	64.1	100
10	3600	56.9	98.3	42.4	100	3600	51.1	97.5	47.7	100	3600	46.1	94.9	51.1	100
12	3600	58.8	97.9	40.3	90	3600	60.0	97.3	38.9	80	3600	53.2	96.2	45.1	90
14	3600	58.1	98.2	41.1	100	3600	53.1	98.1	46.0	100	3600	52.6	97.6	46.3	100
16	3600	70.5	98.1	28.9	70	3600	65.5	97.7	33.7	80	3600	70.2	97.3	29.0	70
Avg	3446	51.6	95.3	45.7	94	3092	43.8	93.9	51.7	93	2898	38.3	92.4	56.0	94

an average of 98.5 (as opposed to 100 with both MINLP-I and MINLP-II), MINLP-III yields LBs of substantially better quality. MINLP-I yields an average LB of 5.7, with feasible solutions found only for 7% of the instances. MINLP-II achieves an average LB of 25.4, with feasible solutions found for 37% of the instances. With MINLP-III, we arrive at a much larger average LB of 85.9 (remember that, due to the way the games are constructed, the leader's utility is upper bounded by 100), with feasible solutions found in 97% of the cases.

5.2 Pessimistic case

For the experiments with our branch-and-bound algorithm, we use GUROBI 6.5.1 for the solution of the MILP Subproblem (9), while employing SCIP 3.2.1 for the solution of nodes (S^+, S^-) with $S^+ = \emptyset$ and to perform the feasibility check. The tree search procedure is implemented in Python. To select the next node to process, we adopt a "worst bound" policy. This leads to always exploring a sequence of nodes (S^+, S^-) with $S^- = \emptyset$ until a leaf node is reached, thus quickly obtaining a feasible solution. The results are reported in Table 2 for games with up to $m = 16$ actions, obtained with different values of ϵ, namely, $\epsilon = 0.1, 1, 5$.[5]

The table shows that the algorithm finds solutions of better quality for larger values of ϵ, in spite of the fact that, the larger ϵ, the poorer the solution should be, as a consequence of larger portions of the leader's feasible region Δ_3 being discarded after a branching operation. This result is due to the fact that, with a larger ϵ, fewer nodes are created and, thus, leaf nodes are reached much faster, resulting in more feasible solutions being found in the time limit. Note that, as the size of the games increases and the algorithm becomes less effective, the method seems to be less affected by the choice of ϵ.

From a gap of, on average, 51.6% obtained with $\epsilon = 0.1$, we achieve one of 43.8% with $\epsilon = 1$, and one of 38.3% with $\epsilon = 5$. While the UB is not much affected by ϵ, we register a larger improvement in the LB, which goes from, on average, 45.7 with $\epsilon = 0.1$ to 51.7 with $\epsilon = 1$ to 56.0 with $\epsilon = 5$. As the table shows, the problem becomes much harder to solve for larger values of m, with an average gap which, from the 65.5%-70.5% range for $m = 16$, considering the three values of ϵ, reaches 99% for $m = 18$ (not shown in the table).

Notwithstanding the problem being a nonconvex pessimistic bilevel program, we remark that the branch-and-bound algorithm we proposed manages to find, with $\epsilon = 5$, solutions in 70% of the cases with an average (approximate, due to ϵ) gap of ~70% for games with

[5] We omit the results for larger values of m as, for them, the algorithm often fails to find a feasible solution.

$m = 16$ actions (4096 payoffs in each of the $n = 3$ matrices U_1, U_2, U_3). Such games are not much smaller, in terms of the number of outcomes, than those solved in papers concerned with the computation of an optimal NE in the single level case, see [15], which is a special case of the problem of computing a SLMFNE obtained when x_3 is restricted to a constant.

6 Concluding remarks

We have considered the problem of computing leader-follower equilibria in games with two or more followers, assuming that, after witnessing the leader's commitment to a mixed strategy, the followers play a mixed-strategy Nash equilibrium. We have proposed three mixed-integer nonconvex mathematical programming formulations for the optimistic case, and an *ad hoc* branch-and-bound method for the pessimistic one. Computational experiments have revealed that, with the last of the three formulations for the optimistic case, we can obtain average gaps smaller than 13% for games with up to 45 actions per player while, with our branch-and-bound algorithm for the pessimistic case, we obtain average (approximate) gaps below 70% for games with up to 16 actions. Future work includes the design of primal heuristics to be embedded in the branch-and-bound algorithm for the pessimistic problem, as well as the construction of dual bounds via the adoption of relaxed optimality conditions.

References

1. E. Amaldi, A. Capone, S. Coniglio, and L. G. Gianoli. Network optimization problems subject to max-min fair flow allocation. *IEEE COMMUN LETT*, 17(7):1463–1466, 2013.
2. B. An, J. Pita, E. Shieh, M. Tambe, C. Kiekintveld, and J. Marecki. Guards and Protect: Next generation applications of security games. *ACM SIGecom Exchanges*, 10(1):31–34, 2011.
3. E. Balas. Disjunctive programming: Properties of the convex hull of feasible points. *DISCRETE APPL MATH*, 89(1):3–44, 1998.
4. N. Basilico, S. Coniglio, and N. Gatti. Methods for Finding Leader-Follower Equilibria with Multiple Followers: (Extended Abstract). In *Proc. of AAMAS*, pages 1363–1364. International Foundation for Autonomous Agents and Multiagent Systems, 2016.
5. A. Caprara, M. Carvalho, A. Lodi, and G.J. Woeginger. Bilevel knapsack with interdiction constraints. *INFORMS J COMPUT*, 28(2):319–333, 2016.
6. V. Conitzer and D. Korzhyk. Commitment to correlated strategies. In *Proc. of AAAI*, 2011.
7. V. Conitzer and T. Sandholm. Computing the optimal strategy to commit to. In *ACM EC*, pages 82–90, 2006.
8. V. Conitzer and T. Sandholm. New complexity results about nash equilibria. *GAME ECON BEHAV*, 63(2):621–641, 2008.
9. C. Kiekintveld, M. Jain, J. Tsai, J. Pita, F. Ordóñez, and M. Tambe. Computing optimal randomized resource allocations for massive security games. In *Proc. of AAMAS*, pages 689–696, 2009.
10. M. Labbé and A. Violin. Bilevel programming and price setting problems. *ANN OPER RES*, 240(1):141–169, 2016.
11. S. Leyffer and T. Munson. Solving multi-leader–common-follower games. *OPTIM METHOD SOFTW*, 25(4):601–623, 2010.
12. Z.-Q. Luo, J.-S. Pang, and D. Ralph. *Mathematical programs with equilibrium constraints.* Cambridge University Press, 1996.
13. J. Matuschke, S.T. McCormick, G. Oriolo, B. Peis, and M. Skutella. Protection of flows under targeted attacks. *OPER RES LETT*, 45(1):53–59, 2017.

14 E. Nudelman, J. Wortman, Y. Shoham, and K. Leyton-Brown. Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. In *Proc. of AAMAS*, pages 880–887. IEEE Computer Society, 2004.

15 T. Sandholm, A. Gilpin, and V. Conitzer. Mixed-Integer Programming Methods for Finding Nash Equilibria. In *Proc. of AAAI*, pages 495–501, 2005.

16 Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game Theoretic and Logical Foundations*. Cambridge University Press, 2008.

17 B. von Stengel and S. Zamir. Leadership games with convex strategy sets. *GAME ECON BEHAV*, 69:446–457, 2010.

18 A.B. Zemkoho. Solving ill-posed bilevel programs. *SET-VALUED ANAL*, 24(3):423–448, 2016.

The Impact of Landscape Sparsification on Modelling and Analysis of the Invasion Process*†

Daniyah A. Aloqalaa[1], Jenny A. Hodgson[2], and Prudence W. H. Wong[3]

1 **Department of Computer Science, University of Liverpool, Liverpool, UK**
d.a.aloqalaa@liverpool.ac.uk

2 **Department of Evolution, Ecology and Behaviour, University of Liverpool, Liverpool, UK**
jenny.hodgson@liverpool.ac.uk

3 **Department of Computer Science, University of Liverpool, Liverpool, UK**
pwong@liverpool.ac.uk

Abstract

Climate change is a major threat to species, unless their populations are able to invade and colonise new landscapes of more suitable environment. In this paper, we propose a new model of the invasion process using a tool of landscape network sparsification to efficiently estimate a duration of the process. More specifically, we aim to simplify the structure of large landscapes using the concept of sparsification in order to substantially decrease the time required to compute a good estimate of the invasion time in these landscapes. For this purpose, two different simulation methods have been compared: *full* and *R-local* simulations, which are based on the concept of dense and sparse networks, respectively. These two methods are applied to real heterogeneous landscapes in the United Kingdom to compute the total estimated time to invade landscapes. We examine how the duration of the invasion process is affected by different factors, such as dispersal coefficient, landscape quality and landscape size. Extensive evaluations have been carried out, showing that the R-local method approximates the duration of the invasion process to high accuracy using a substantially reduced computation time.

1998 ACM Subject Classification E. [Data] Graphs and Networks, G.3 Probability and Statistics, J.3 [Life and Medical Sciences] Biology and genetics

Keywords and phrases Landscape sparsification, invasion process, network sparsification, dense and sparse networks

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.32

1 Introduction

Climate change and land use change are two threats that cause the extinction of numerous species [4, 7, 12, 17]. It was observed that species are responding to climate change by shifting their geographical area [1, 16], however the ability of their population to shift depends on the availability of suitable habitat to shift and colonise [4, 6, 12, 13]. Species become under a high risk of extinction if they are shifting very slowly or do not have the ability to shift [15]. Therefore, to maintain the functioning of an ecosystem and services in a changing

* The source codes and data used are available at https://github.com/Aloqalaa/Landscape-Sparsification-on-Modelling-Invasion-Process.

† Jenny A. Hodgson acknowledges support from NERC grant ref NE/L002787/1.

© Daniyah A. Aloqalaa, Jenny A. Hodgson, and Prudence W. H. Wong;
licensed under Creative Commons License CC-BY

16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 32; pp. 32:1–32:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

climate, it becomes an important need to facilitate the adaptation of species, especially by enabling them to shift to new locations with more suitable climate [5]. It is an urgent need for policymakers and nature conservation organisations to find out whether and how they can facilitate range shifts [5]. A number of empirical and theoretical studies have shown that spatial arrangement of habitats is an important factor that affects the speed of advance to new landscapes with more suitable climate [3, 5, 17]. Hodgson et al. [5] found the evidence of the benefits of using different tools such as habitat corridors and stepping stones to speed up shifting. However, it is still difficult for conservationists to make decisions that can facilitate range shifts in large landscapes, therefore there is a need for a tool efficiently computing the invasion time of the original and modified landscapes. Minimising computation time is especially important because the ultimate aim is for a decision-making tool that can tune the arrangement of the modified landscape to find scenarios where a small addition of habitat leads to a large decrease in invasion time. Running many scenarios with different permutations of habitat could require excessive computation times even with moderately-sized landscapes. Furthermore, planning for climate change requires the consideration of large landscapes (e.g., temperature isoclines are expected to shift at several km per decade).

In this study, our focus is to build a new sparse computational model for the invasion process using the network modelling approach. To model invasion process in a given landscape, we create a landscape network, where each vertex represents a patch of habitat (henceforth patch) in the landscape. We distinguish two sets of patches: the source patches represent initially populated patches in which species are located, and the target patches represent the target locations for the invasion process. The invasion process is to populate (some) target patches and we aim to estimate the time needed for achieving this. A stochastic model from [5] has been implemented in the simulation, which is based on the probability of a patch to be invaded expressed by a formula depending on various characteristics (distance, quality of patch, etc.) of all other patches in the network. Such simulation is computationally expensive, especially when the number of patches is large.

In this paper we propose to approximate the computed invasion time by exploiting network sparsification. We call the original approach in [5] the *full* invasion simulation method. In the protocol implementing *full* invasion method, in each round of the invasion process each populated patch tries to populate (independently) all other unpopulated patches in the landscape. On the other hand, we propose the *R-local* invasion method, and associated protocol, such that in each round, every patch only tries to populate other unpopulated patches within a local distance R. The *full* invasion protocol can be seen as an *R-local* invasion protocol with R equal to the diameter of the landscape. With a smaller value of R, the *R-local* invasion process is expected to take less time in each round of computation, while it may take more rounds to populate the target patches. If the local distance R is chosen properly, the *R-local* invasion protocol can compute a comparable (to the *full* method) duration of invasion process, while the computation time can be substantially reduced. That means our proposed method is much more efficient (less computational time) than the existing model. One important characteristics to consider when computing the total time of experiment is also the number of simulations needed for the (average) invasion time to stabilise on the outputted duration of invasion. In this paper, we illustrate how to determine the local distance R systematically to reach a good trade-off between quality of estimate and computation time.

In both *full* and *R-local* protocols, we investigate the effect of three factors: species' mean dispersal distance, landscape quality and landscape size, on the invasion duration. We apply the *full* and *R-local* protocols to real heterogeneous landscapes in the United Kingdom. An

Table 1 Aggregate classes [10].

Aggregate class number	Aggregate class	Aggregate class number	Aggregate class
1	Broadleaf woodland	6	Mountain, heath, bog
2	Coniferous woodland	7	Saltwater
3	Arable	8	Freshwater
4	Improved grassland	9	Coastal
5	Semi-natural grassland	10	Built-up areas and gardens

extensive experimental evaluation with real data illustrates the effectiveness and accurateness of the proposed *R-local* protocol in estimating the duration of invasion process in large landscapes in a relatively short time, with respect to the previously used *full* method.

Technically speaking, the work presented in this paper combines ideas from probability and random processes [2, 8, 9] with some use of network/graph foundations [11].

2 Materials and methods

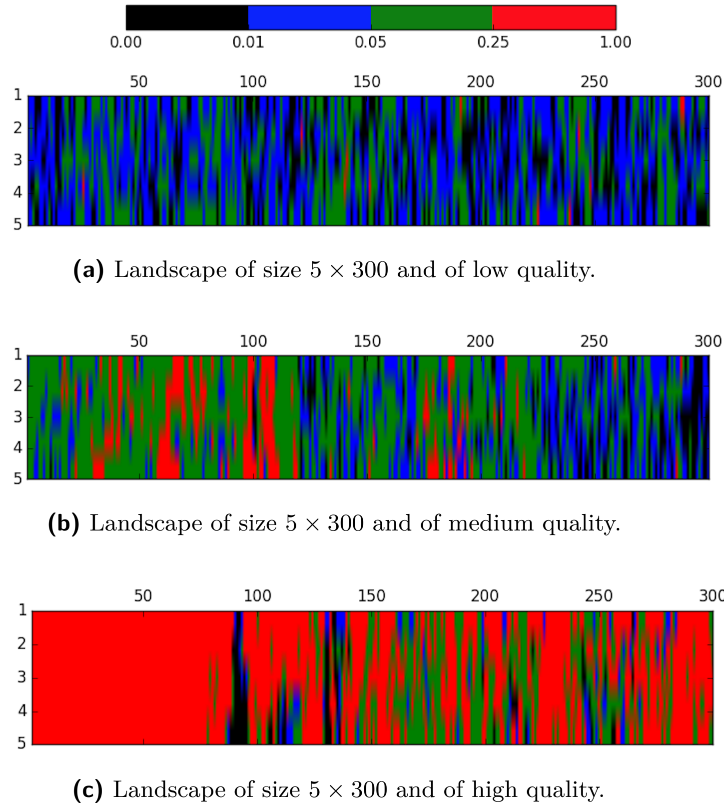
2.1 Notation and technical preliminaries

We are given a 2-dimensional rectangular grid landscape as an input, which we represent as a landscape graph $G = (V, E)$. Denote by $n = |V|$ the number of vertices/patches in the landscape graph, and by $m = |E|$ the total number of edges between patches. Let $q(v)$ denote the quality of a patch v, where the quality is a number between zero and one given as an input. We distinguish two sets of patches, S and T, where S denotes the set of populated source patches that are non-zeros in quality and T denotes the set of unpopulated target patches that are also non-zeros in quality. In addition, we define the maximum and minimum quality as $q_{max} = \max\{q(v) : v \in V\}$ and $q_{min} = \min\{q(v) > 0 : v \in V\}$, respectively. We define the height H as the difference in y-coordinates of any top and bottom patch in the landscape graph G and the width W as the difference in x-coordinates of any left patch in the source set and right patch in the target set. We also define *R-local* graph $Loc(G, R) = (V', E')$ as a subgraph of G that contains all the vertices of the landscape graph G (i.e., $V' = V$) and a set of edges $E' \subseteq E$ such that for any $(u, v) \in E$, $(u, v) \in E'$ if the Euclidean distance between vertices v and u is at most R. For a defined *R-local* graph $Loc(G, R)$, we define d_{min} as the minimum distance d such that $Loc(G, d)$ connects S to T, i.e., d_{min} is the minimum distance d such that every node in T is reachable from some node in S in graph $Loc(G, d)$.

2.2 The studied landscape

For evaluation purposes, the dataset of the 1km resolution raster version of the Land Cover Map 2007 (LCM2007) for Great Britain [14] is used.

To examine the proposed methods, different sized landscapes from different maps of the aggregate classes are extracted. The aggregate classes data contain one tiff file for each land use class as in Table 1. Each map consists of 1300 rows/height (pixels) and 700 columns/width (pixels) and each 1km pixel provides the percentage cover of a particular land cover at LCM2007 Class level [10]. We consider the percentage cover at each patch in such an extracted landscape as the quality of each patch. For examination purposes, from percentage values we formed three groups of landscape qualities, namely: low quality, medium quality and high quality to represent the quality of the extracted landscape. If the average of all



(a) Landscape of size 5 × 300 and of low quality.

(b) Landscape of size 5 × 300 and of medium quality.

(c) Landscape of size 5 × 300 and of high quality.

Figure 1 The studied landscapes; three landscapes of size 5 × 300 and of low, medium and high quality extracted from LCM2007 UK (aggregate classes) maps. In each landscapes, the colour corresponds to the quality of each patch; black, blue, green, and red corresponds to zero, low (0.01-0.05), medium (0.05-0.25), and high quality (0.25-1), respectively.

patches qualities in such an extracted landscape is between 0% and 5%, 5% and 25%, 25% and 100%, then the extracted landscape is of low quality, medium quality, and high quality, respectively. For each quality type, we extract a rectangular landscape that consists of 5 rows and 300 columns. Landscapes of low and medium qualities were extracted from semi-natural grassland UK map, while the one of high quality from an improved grassland UK map. On these extracted rectangular landscapes, we assume that all patches at the first column of each landscape are occupied and the goal is to populate patches at the target columns (col. 10, 20, 30, etc.). Each landscape is extracted according to the following criteria:

1. The qualities of all source occupied patches are non-zeros.
2. At each target column, at least one of the patches is non-zero in quality.

Figure 1 shows real landscapes used which are extracted from two LCM2007 UK (aggregate classes) maps.

2.3 New formulas estimating duration of the invasion process

For a given landscape graph G, we use the formula of colonisation probability proposed by Hodgson et al. [5] to define the *transition probability* between patches v and u as $p(v,u) = q(v) \cdot \frac{\exp(-\alpha d(v,u))}{(\frac{2\pi}{\alpha^2})-1}$, where $\alpha > 0$ is the dispersal coefficient assumed to be the same for all patches and $d(v,u)$ is the Euclidean distance between patches v and u.

For a given landscape graph G, source S and target T, we define "all non-zero target patches" as the total number of target patches that are non-zeros in quality, and "majority of all non-zero target patches" as the number of patches being more than half of the total number of non-zero quality target patches. In this work, we consider three types of invasions, namely: *first* success, *majority* success and *all* successes invasion. The *first* success invasion measures the estimated time to populate any of the non-zero target patches. Therefore, the *first* success is defined as the estimated time needed until the first non-zero target patch becomes populated. The *majority* success is the estimated time to populate the majority (more than 50%) of all non-zero target patches. Finally, *all* successes is defined as the estimated time to populate all non-zero target patches.

Based on the formula of the *transition probability*, we propose three new estimating times of invasions in such a landscape. For a given landscape graph G, in order to hop from source S to target T, we need to find a path that contains at most $\frac{H+W}{d_{min}(G)}$ hops. According to the formula of *transition probability* between patches, the probability of a single hop in the landscape graph G is at least $q_{min}(G) \cdot \frac{\exp(-\alpha \cdot d_{min}(G))}{\left(\frac{2\pi}{\alpha^2}\right)-1}$. Thus, the expected time for each hop is the inverse of its probability, i.e., $\frac{\left(\frac{2\pi}{\alpha^2}\right)-1}{q_{min}(G)\cdot\exp(-\alpha\cdot d_{min}(G))}$. Since the number of hops that are needed in such a path to connect source S with target T is $\frac{H+W}{d_{min}(G)}$, the total expected time of invasion for the *first* success is at most

$$\frac{H+W}{d_{min}(G)} \cdot \frac{\left(\frac{2\pi}{\alpha^2}\right)-1}{q_{min}(G)\cdot\exp\left(-\alpha\cdot d_{min}(G)\right)} \cdot c \ ,$$

where c is a small constant to be determined by simulation. Consequently, the total expected time of invasion for the *majority* success is at most

$$\left[\frac{H+W}{d_{min}(G)} \cdot \frac{\left(\frac{2\pi}{\alpha^2}\right)-1}{q_{min}(G)\cdot\exp\left(-\alpha\cdot d_{min}(G)\right)} + \left(\frac{H}{2}-1\right) \cdot \frac{\left(\frac{2\pi}{\alpha^2}\right)-1}{q_{min}(G)\cdot\exp\left(-\alpha\cdot d_{min}(G)\right)}\right] \cdot c \ ,$$

where $\left(\frac{H}{2}-1\right)$ is an upper bound on the total number of *majority* of target patches decreased by one and c is again a small constant to be interpolated by simulation. Therefore, the total expected time of invasion for *all* successes is at most

$$\left[\frac{H+W}{d_{min}(G)} \cdot \frac{\left(\frac{2\pi}{\alpha^2}\right)-1}{q_{min}(G)\cdot\exp\left(-\alpha\cdot d_{min}(G)\right)} + (H-1) \cdot \frac{\left(\frac{2\pi}{\alpha^2}\right)-1}{q_{min}(G)\cdot\exp\left(-\alpha\cdot d_{min}(G)\right)}\right] \cdot c \ ,$$

where $(H-1)$ is an upper bound on the total number of *all* target patches decreased by one and c is a small constant to be interpolated by simulation.

2.4 The *R-local* simulation method

We simulate the behaviour of the invasion process by building a simulator that uses the *R-local* method to compute the number of rounds needed for invasion. The inputs to the simulator are: a two-dimensional array that represents a given real landscape and stores qualities of patches, a source vector containing indices of populated patches, a target vector containing indices of unpopulated target patches, and dispersal coefficient α (in Algorithm 1 initialised to 0.25). For a given landscape, the simulator constructs a two-dimensional array of a size equal to the one of the given landscape. Each cell in the constructed array corresponds to a patch in the landscape and can take only two values, zero or one, where zero means the cell is unpopulated while one means it is populated. At the beginning of the invasion process, only source populated patches take value of one and others take value of zero. The

simulator returns an estimated duration needed (number of rounds) to invade targets by the use of real probabilities for each pair of patches v, u, in which v is populated and u is not. We use transition probabilities between patches v and u to decide whether patch v populates patch u or not.

More formal description of Algorithm 1 provides the structure of the *R-local* method. The generic structure of the *R-local* method contains inputs (as mentioned above), outputs, and COUNT ROUNDS function. The COUNT ROUNDS function counts the number of rounds required for *first*, *majority* and *all* successes and to compute the real time execution for each simulation. The function includes nested loops of three levels. The main loop (lines 12–37) counts the number of rounds to populate target patches. The second level loop (lines 14–36) is for all populated patches that are trying to populate unpopulated patches. The inner level loop (lines 17–33) is for all unpopulated patches. Each unpopulated patch becomes populated if the *transition probability* between the populated and unpopulated patches is greater than a random generated number between zero and one (lines 22-25). We consider only populating a patch with non-zero quality. Each time when an unpopulated target patch becomes populated, the algorithm checks if the total number of non-zero patches at target is equal to one or *majority* or *all* targets' number, and the number of rounds is recorded accordingly. The COUNT ROUNDS function terminates when all non-zero target patches become populated and returns the number of rounds needed for each type of successful invasions as well as the execution time of simulation.

Recall that in the *full* invasion method, each populated patch in the landscape tries to populate every other unpopulated patch in the whole landscape, while in the *R-local* invasion method each populated patch in the landscape only tries to populate every other unpopulated patch within a local distance R around the populated patch. Thus, parameter R in the *full* method takes the whole size (diameter) of the landscape (i.e., $R = H + W$), while in the *R-local* method we propose an equation to compute the local distance R. In addition, the inner level loop in COUNT ROUNDS function runs for all unpopulated patches in the whole landscape (*from 0 to* $H + W$) in the *full* method, while runs only for unpopulated patches that are of distance at most R from the populated patch (*from populated patch to* R) in the *R-local* method (lines 17 and 18 in Algorithm 1).

It is expected that the number of rounds required for the *R-local* method is larger. In our simulation, we aim to find the local distance R that allows the following accuracy:

$$\frac{\text{FULL}}{\text{R-LOCAL}} = \frac{\text{average number of rounds using the } \textit{full} \text{ method}}{\text{average number of rounds using the } \textit{R-local} \text{ method}} \geq 90\% \ .$$

As a starting point, we run simulation using both *full* and *R-local* methods with some expected local distances R. It has been observed that the required local distance R for the FULL/R-LOCAL ratio to be at least 90% in the *first* success is greater than or equal to the needed local distance R for *majority* and *all* successes. Based on this observation, we propose an equation that computes the local distance R for a given landscape graph G based on the total expected time of invasion for the *first* success. Observe that the probability of a single hop in a given landscape graph G is less than the inverse of the total expected time of the invasion process for the *first* success:

$$q_{min}(G) \cdot \frac{\exp\left(-\alpha \cdot d_{min}(G)\right)}{\left(\frac{2\pi}{\alpha^2}\right) - 1} \ll \frac{1}{\text{total expected time of invasion for } \textit{first} \text{ success}} \ .$$

Therefore, the following holds:

$$q_{max}(G) \cdot \frac{\exp\left(-\alpha \cdot R(G)\right)}{\left(\frac{2\pi}{\alpha^2}\right) - 1} \ll \frac{d_{min}(G)}{H + W} \cdot q_{min}(G) \cdot \frac{\exp\left(-\alpha \cdot d_{min}(G)\right)}{\left(\frac{2\pi}{\alpha^2}\right) - 1}$$

$$-\alpha\left(R(G)-d_{min}(G)\right)<\ln\left(\frac{d_{min}(G)\cdot\bar{q}(G)}{H+W}\right).$$

Finally, we get the best (smallest) local distance R needed to create a landscape subgraph $Loc(G,R)$, for a given landscape graph G, and guarantees the FULL/R-LOCAL accuracy is:

$$R(G)\approx\left[\frac{1}{\alpha}\cdot\ln\left(\frac{H+W}{d_{min}(G)\cdot\bar{q}(G)}\right)+d_{min}(G)\right]\cdot c\,, \tag{1}$$

where $\bar{q}(G)=\frac{q_{min}(G)}{q_{max}(G)}$ and c is a small constant to be determined by simulation.

2.4.1 Interpolating constant c in Equation (1)

We use the following three objective functions in order to interpolate constant c in Equation (1). We use terminology "approx R" and "opt R" to express the local distance R using Equation (1) and simulation such that it is the smallest distance satisfies the FULL/R-LOCAL accuracy, respectively. We call it opt R because this value of R fulfills our goal of accuracy.

1. The Euclidean distance objective function (ED) chooses the constant c such that it minimises the sum over all prefixes z of the square difference between the approx R and opt R:

$$c_{ED}=\underset{c\in\mathbb{R}^{+}}{\operatorname{argmin}}\left\{\sum_{j=1}^{z}(approxR_j\cdot c-optR_j)^2\right\}.$$

2. The absolute objective function (AB) chooses the constant c such that it minimises the sum over all prefixes z of the absolute difference between the approx R and opt R:

$$c_{AB}=\underset{c\in\mathbb{R}^{+}}{\operatorname{argmin}}\left\{\sum_{j=1}^{z}|approxR_j\cdot c-optR_j|\right\}.$$

3. The min-max (MM) objective function chooses the constant c such that it minimises the approx R to be greater than or equal to opt R for all prefixes z:

$$c_{MM}=\underset{c\in\mathbb{R}^{+}}{\operatorname{argmin}}\left\{\max_{1\leq j\leq z}(approxR_j\cdot c-optR_j)\geq 0\right\}.$$

2.5 Simulation

2.5.1 Main simulation

The simulation is directed at four goals. The first is to monitor the behaviour of *full* and *R-local* methods and compare results obtained by each method. The second is to investigate what values of local distance R would allow the FULL/R-LOCAL accuracy. Based on the results obtained from the simulation, the third goal is to predict an equation for the local distance R, depending on landscape size, dispersal coefficient α and landscape quality. Finally, we aim to combine results from simulation and the predicted equation to compare them and conduct an independent validation based on the value obtained from the proposed equation on the local distance R.

For each prefix $5\times 10, 5\times 20, 5\times 30, ..., 5\times 300$ in each of the extracted rectangular landscapes in Figure 1, we run the *full* simulation and, simultaneously, the *R-local* simulation

Algorithm 1 Modelling Invasion Process using *R-local* Method.

```
1: Inputs:
      2-dimensional array G, which stores qualities of patches in a real landscape
      Source: vector S contains indices of initial populated patches
      Target: vector T contains indices of unpopulated target patches
      α ← 0.25
2: Outputs:
      Number of rounds needed for first, majority and all successful invasions
      Execution time of simulation
3: function COUNT ROUNDS(G, S, T, α)
4:     SimulationStartTime ← record start time of simulation
5:     R ← Use Equation (1)
6:     Create 2-dimensional array B having size equal to array G ← 0
7:     for each index of populated patch in vector S do
8:         B(index) ← 1
9:     end for
10:    Counter ← 0                    ▷ counter to count rounds for successful invasions
11:    StartTime ← Record start time of rounds
12:    while any of target patch in T is unpopulated do
13:        Counter ← Counter+1
14:        for i ← 0 to number of rows in array G do        ▷ loop for all populated patches (1)
15:            for j ← 0 to number of columns in array G do
16:                if patch B(i, j) is populated then
17:                    for k ← i − R to i + R do              ▷ loop for all unpopulated patches (0)
18:                        for l ← j − R to j + R do
19:                            if patch B(k, l) is unpopulated and its quality ≠ 0 then
20:                                distance ← Euclidean distance between B(i, j) and B(k, l)
21:                                if 0 < distance ≤ R then
22:                                    p ← Transition probability between B(i, j) and B(k, l)
23:                                    w ← Generate a random number between 0 and 1
24:                                    if w < p then
25:                                        Populate patch B(k, l)
26:                                        if populated patch B(k, l) is at target column then
27:                                            Check the invasion type first or majority or all
28:                                        end if
29:                                    end if
30:                                end if
31:                            end if
32:                        end for
33:                    end for
34:                end if
35:            end for
36:        end for
37:    end while
38:    SimulationEndTime ← Record end time of simulation
39:    SimulationTimeExecution ← SimulationEndTime−SimulationStartTime
40:    return Counter, SimulationTimeExecution
41: end function
```

with some predicted distances; ideally, one of these R gives the FULL/R-LOCAL accuracy, while the local distance R decreased by one does not satisfy it. We consider four values for the dispersal coefficient α: 0.25, 0.5, 1 and 2. For each prefix and for each dispersal coefficient α, we run *full* and *R-local* simulation 100 times and compute the average number of rounds for *first*, *majority*, and *all* successes.

2.5.2 Validation of the *R-local* simulation method

We are interested in the most convenient time for stopping or cutting simulation and determine the stabilisation time (ST) of simulation. For that purpose we define the stabilisation time (ST) for a given landscape as the time t such that the change in average number of rounds for *all* successes (AFAS) between t and $2t$ is less than or equal to 2%: $\forall \tau \in (t, 2t]$, $|AFAS(\tau) - AFAS(t)| \leq 0.02 \cdot AFAS(t)$, where $AFAS(\tau) = \frac{\sum_{j=1}^{\tau} AS(j)}{\tau}$ and $AS(j)$ is the number of rounds needed for the *all* successes at experiment j.

To test the robustness of our method, we have used different sizes from the sizes used in deriving R. We apply it to four different sizes of landscapes: 5×50, 10×50, 15×50 and 20×50, as in the following steps.

1. Extract randomly three landscapes of each size from different LCM2007 UK (aggregate classes) maps for each quality low, medium, and high.
2. On these extracted landscapes and for each considered value of the dispersal coefficient α:
   a. Run *full* simulation, stop simulation at the ST and record the results.
   b. Compute the average number of rounds for *first*, *majority* and *all* successes, and the average of the execution times of *full* simulation.
   c. Compute the minimum quality q_{min}, the maximum quality q_{max} and the minimum distance d_{min}.
   d. Compute three local distances R using Equation (1) with the constants in Table 3 which are calculated as in Section 2.5.1 with the data in Section 2.4.1 (see Section 3).
   e. Run *R-local* simulation with each value of the three computed local distances R (based on c_{ED}, c_{AB}, and c_{MM} constants), stop simulation at the ST and record the results.
   f. Compute the average number of rounds for *first*, *majority* and *all* successes, and the average of the execution times of *R-local* simulation.
   g. For all three types of successful invasions and all local distances R computed in (d), compute the FULL/R-LOCAL accuracy. Then, check all accuracies if they are guaranteed.
   h. Specify which of the computed local distance R is the opt, where the opt one is the smallest distance that gives the FULL/R-LOCAL accuracy to be at least 90%.
   i. Compute the ratio between the average of the execution times (AETS) of *full* and *R-local* simulation methods.

3 Results

3.1 Main simulation

The estimated time of invasion (i.e., average number of rounds over 100 independent experiments) has been computed for each prefix 5×10, 5×20, 5×30, ..., 5×300 in each landscape of low, medium, and high quality using *full* and *R-local* simulation methods. In order to investigate the local distance R that allows FULL/R-LOCAL accuracy in each prefix in each of the extracted landscapes we use some predicted local distances R to run

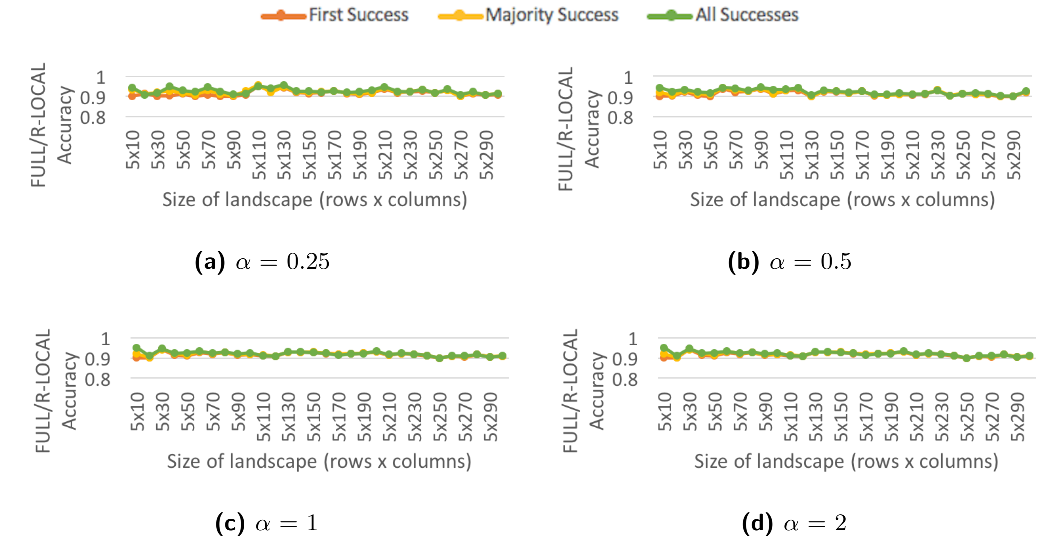


Figure 2 The FULL/R-LOCAL accuracy for each prefix in landscape of size 5×300 and of **low** quality when the dispersal coefficient α takes values of 0.25, 0.5, 1 and 2.

R-local simulation; ideally, one of the predicted distances is the opt that guarantees the FULL/R-LOCAL accuracy, while decreasing the opt local distance R by one does not satisfy it. The simulation results show that the FULL/R-LOCAL accuracy of at least 90% is satisfied in all scenarios, as shown in Figures 2-4. On these landscapes, the opt local distances R used for simulations and guaranteeing the FULL/R-LOCAL accuracy are provided in Figure 5a, while Figure 5b shows the approx local distances R using Equation (1). From the opt local distances R in Figure 5a, we observe that the dispersal coefficient α is the most important parameter in both simulation methods. In all qualities, it has been investigated that a larger local distance R is required when the dispersal coefficient α equals to 0.25, while for 0.5, 1 and 2 a smaller local distance R is sufficient. Therefore, the local distance R that ensures the FULL/R-LOCAL accuracy depends on the dispersal coefficient α, and hence with decreasing mean dispersal distance: R decreases with the increase in α. Furthermore, a logarithmic growth has been observed in the local distance R with the growth of landscape size. On the other hand, the difference in the landscape quality has not caused a significant difference in the local distance R.

Comparison of the obtained results based on the proposed equation with simulation results demonstrates that the proposed equation (Equation (1)) gives a good estimate of the local distance R. We define the error rate between the approx and opt local distances R as:

$$\frac{\sum_{j=1}^{z}(approxR_j - optR_j)}{\sum_{j=1}^{z} optR_j}.$$

Table 2 gives the error rates between the approx and opt local distances R for each 5×300 landscape of low, medium, and high quality. All error rates in Table 2 are high, which means that we need to find the best constant c such that it minimises the error for various dispersal coefficients α and different qualities. Table 3 presents the interpolated constants c based on the objective functions ED, AB and MM for each 5×300 landscape of low, medium, and high quality. These constants are affected by the opt local distances R for all prefixes in each landscape of different quality.

(a) $\alpha = 0.25$ **(b)** $\alpha = 0.5$

(c) $\alpha = 1$ **(d)** $\alpha = 2$

Figure 3 The FULL/R-LOCAL accuracy for each prefix in landscape of size 5×300 and of **medium** quality when the dispersal coefficient α takes values of 0.25, 0.5, 1 and 2.

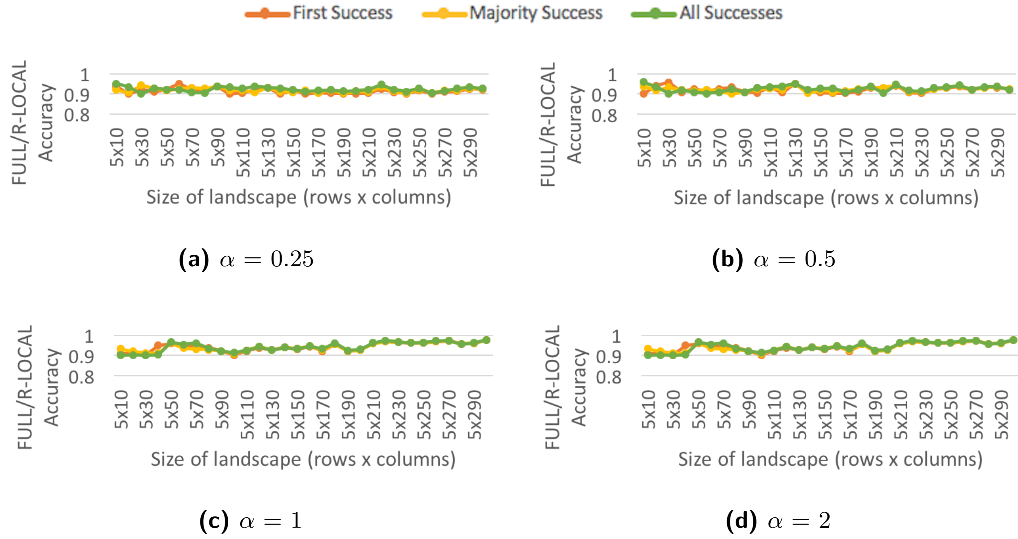
Table 2 Error rate between approx (when constant $c = 1$) and opt local distances R for each 5×300 landscape of low, medium and high quality when the dispersal coefficient $\alpha = 0.25$, 0.5, 1, 2.

Landscape quality	α=0.25	α=0.5	α=1	α=2
Low quality	0.81	0.74	0.73	0.57
Medium quality	0.83	0.86	0.90	1.01
High quality	0.89	0.75	0.56	0.63

In addition, we define the following three error rates, which correspond to constant c produced by each objective function:

1. error rate$(c_{ED}) = \dfrac{\sqrt{\sum_{j=1}^{z} (approxR_j \cdot c_{ED} - optR_j)^2}}{\sum_{j=1}^{z} optR_j}$,
2. error rate$(c_{AB}) = \dfrac{\sum_{j=1}^{z} |approxR_j \cdot c_{AB} - optR_j|}{\sum_{j=1}^{z} optR_j}$, and
3. error rate$(c_{MM}) = \dfrac{\sum_{j=1}^{z} (approxR_j \cdot c_{MM} - optR_j)}{\sum_{j=1}^{z} optR_j}$.

Observe that the error rate gives a measure of how well the constant c interpolated by the corresponding objective function minimises the error rate between the approx and opt local distances R, for a given landscape. When the approx local distance R is very close or equal to the opt local distance R, the error will be small or zero. Table 4 provides the computed error rates between the approx and opt local distances R for each 5×300 landscape of low, medium and high quality, when the constant c is equal to the interpolated c_{ED}, c_{AB}, and c_{MM} (constants in Table 3). As can be seen in Table 4, all error rates are between 0.01



(a) $\alpha = 0.25$ **(b)** $\alpha = 0.5$

(c) $\alpha = 1$ **(d)** $\alpha = 2$

Figure 4 The FULL/R-LOCAL accuracy for each prefix in landscape of size 5×300 and of **high** quality when the dispersal coefficient α takes values of 0.25, 0.5, 1 and 2.

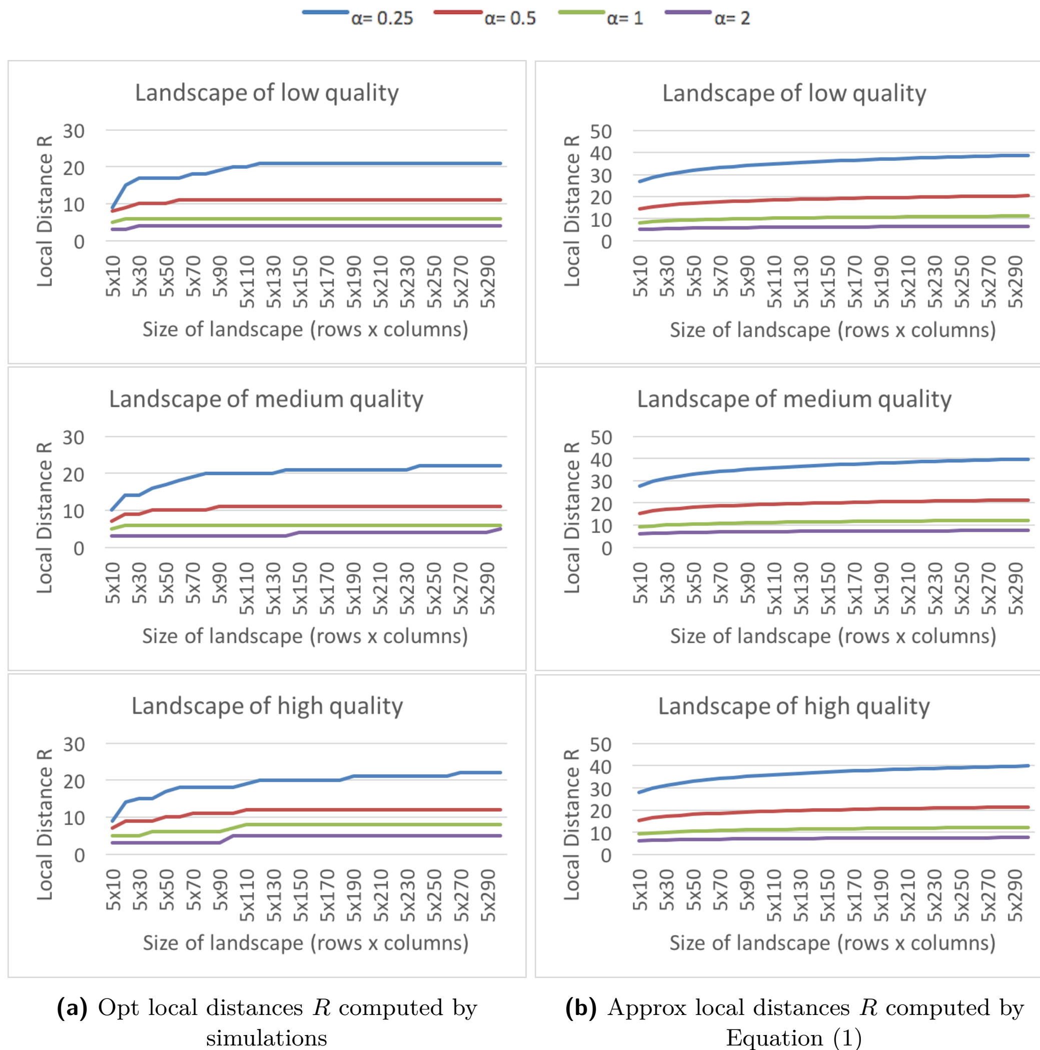
Table 3 The computed constant c by the objective functions ED, AB and MM for each 5×300 landscape of low, medium and high quality when the dispersal coefficient $\alpha = 0.25, 0.5, 1, 2$.

Landscape quality	Parameters	Constant c	α=0.25	α=0.5	α=1	α=2
Low quality	$q_{min} = 0.01$	c_{ED}	0.55	0.55	0.6	0.65
	$q_{max} = 0.64$	c_{AB}	0.55	0.55	0.55	0.65
	$d_{min} = 2$	c_{MM}	0.6	0.65	0.7	0.75
Medium quality	$q_{min} = 0.01$	c_{ED}	0.55	0.55	0.55	0.5
	$q_{max} = 0.96$	c_{AB}	0.55	0.55	0.5	0.5
	$d_{min} = 3$	c_{MM}	0.6	0.6	0.65	0.7
High quality	$q_{min} = 0.01$	c_{ED}	0.55	0.55	0.65	0.6
	$q_{max} = 0.99$	c_{AB}	0.55	0.6	0.65	0.65
	$d_{min} = 3$	c_{MM}	0.6	0.65	0.75	0.75

and 0.4. While c_{ED} and c_{AB} produce error rates smaller than c_{MM}, constant c_{MM} is the best among the three. One of the reasons could be that constant c_{MM} in all landscapes reduces the approx local distance R to be greater than or equal to the opt local distance R. On the other hand, in some cases c_{ED} and c_{AB} decrease the approx local distance R to be less than the opt local distance R, and that means they give an estimated local distance R which does not allow the sought FULL/R-LOCAL accuracy.

3.2 Validation

We performed validation on 36 different landscapes. The 36 landscapes are divided into groups of nine landscapes and the four groups each has size: 5×50, 10×50, 15×50, and 20×50, respectively. For each landscape size, the nine landscapes are further divided into subgroups of three and each group has associated low, medium and high quality, respectively. All landscapes are extracted randomly from different LCM2007 UK (aggregate classes) maps (i.e., aggregate classes in Table 1). On those landscapes, we run *full* and *R-local* simulations independently as described in Section 2.5.2 in order to get the averages of the estimated time



(a) Opt local distances R computed by simulations

(b) Approx local distances R computed by Equation (1)

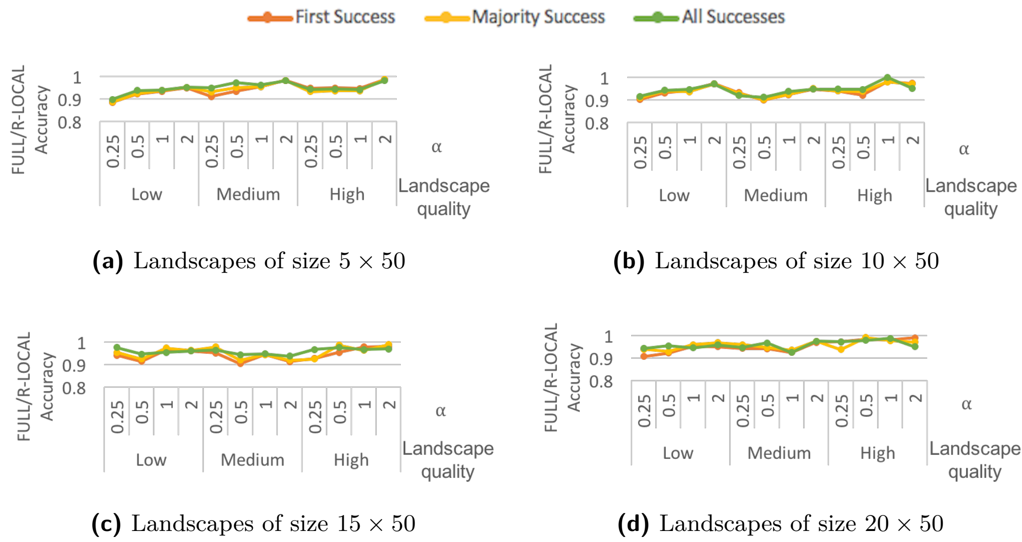
Figure 5 The local distance R for each prefix in landscape of size 5×300 and of low, medium, and high quality when the dispersal coefficient α takes values of 0.25, 0.5, 1 and 2. **(a)**The opt local distances R that allows the FULL/R-LOCAL accuracy and computed by simulations. **(b)**The computed local distances R by the proposed formula (Equation (1), when constant $c = 1$).

of invasion for *first*, *majority* and *all* successes as well as the averages of the execution times of simulations. Running *full* and *R-local* simulations on those landscapes, using constants in Table 3, gives a good result as the FULL/R-LOCAL accuracy has been achieved as presented in Figure 6. It has been illustrated by the validation experiments that the MM objective function is the best function to be used among the three objective functions because it gives constant c_{MM} such that it reduces the approx local distance R to be greater than or equal to the opt for all 36 landscapes.

Furthermore, the validation experiments demonstrate that the total time (TT) of simulation execution, where TT = ST · AETS, needed to compute the estimated duration of the invasion process is substantially reduced by the *R-local* simulation method for all landscapes of different qualities. Figure 7 illustrates how much the *R-local* method is faster than the *full* method for all three qualities. For many cases, the *full* method takes 5-10 times longer to compute and this ratio can become as high as 75 for low quality landscape. We note

Table 4 The error rate between approx and opt local distances R, when $c = c_{ED}, c_{AB}, c_{MM}$, for each 5×300 landscape of low, medium and high quality when α = 0.25, 0.5, 1, 2.

Landscape quality	Parameters	Error rate	α=0.25	α=0.5	α=1	α=2
Low quality	$q_{min} = 0.01$	Error rate (c_{ED})	0.01	0.01	0.01	0.01
	$q_{max} = 0.64$	Error rate (c_{AB})	0.04	0.05	0.06	0.05
	$d_{min} = 2$	Error rate (c_{MM})	0.08	0.12	0.21	0.17
Medium quality	$q_{min} = 0.01$	Error rate (c_{ED})	0.01	0.01	0.01	0.02
	$q_{max} = 0.96$	Error rate (c_{AB})	0.04	0.04	0.05	0.11
	$d_{min} = 3$	Error rate (c_{MM})	0.1	0.11	0.23	0.40
High quality	$q_{min} = 0.01$	Error rate (c_{ED})	0.01	0.01	0.02	0.03
	$q_{max} = 0.99$	Error rate (c_{AB})	0.05	0.05	0.08	0.12
	$d_{min} = 3$	Error rate (c_{MM})	0.13	0.13	0.17	0.22



(a) Landscapes of size 5×50 **(b)** Landscapes of size 10×50

(c) Landscapes of size 15×50 **(d)** Landscapes of size 20×50

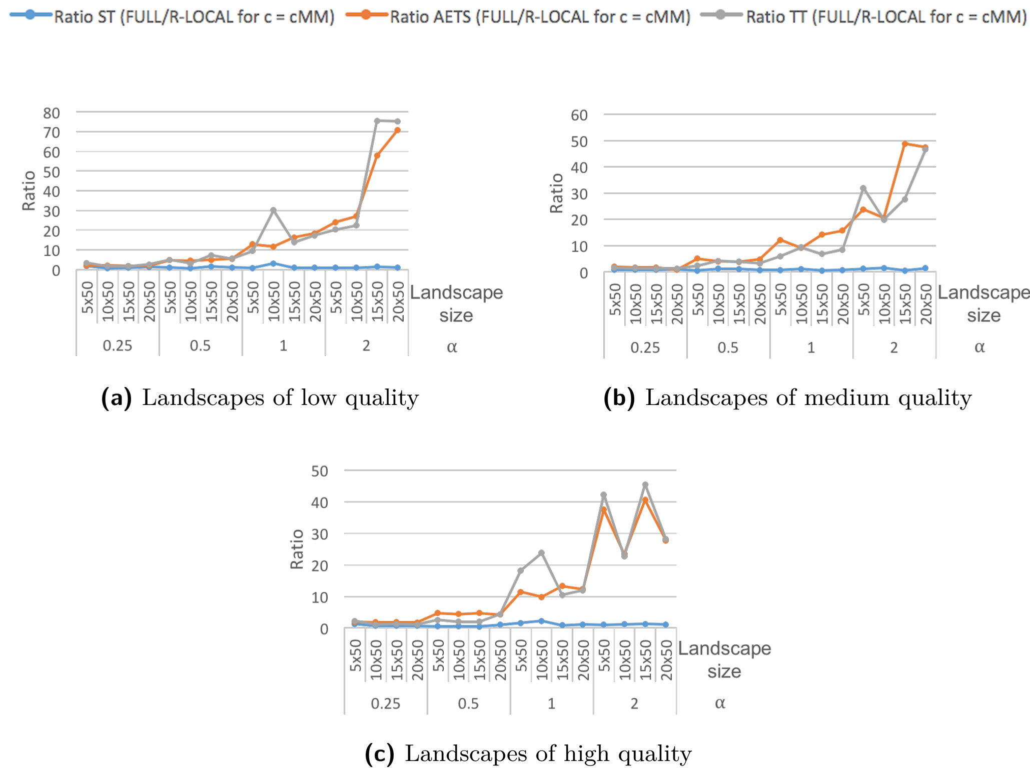
Figure 6 The average of FULL/R-LOCAL accuracy over the three landscapes in each subgroup with same size and same landscape quality; the R-LOCAL is with $R = approxR$ for $c = c_{MM}$. This is done for different values of dispersal coefficients α.

that in general, for a given landscape size, the speedup of the *R-local* method increases as the dispersal coefficient α increases. On the other hand, in most cases, when the dispersal coefficient α is fixed, the speedup increases as the size of landscape increases.

In more details, in the landscape of size 20×50 and of low quality, the average execution time of *full* simulation is 176.9 seconds while only 2.5 seconds in the *R-local* simulation. That means the *full* method takes around 70 times longer to compute. We could envisage that when we are running *full* simulation in very large landscapes e.g., landscape of size 500×500, the computation time will be substantially reduced from maybe weeks/days to hours.

4 Conclusion

This study was prompted by a desire to construct the *R-local* model that visualises the invasion process based on the landscape network sparsification tool to efficiently estimate a duration of the process. The capability of our model is to reduce the time needed to compute the estimated duration of the invasion process on large landscapes while maintaining a comparable duration of the invasion.



(a) Landscapes of low quality

(b) Landscapes of medium quality

(c) Landscapes of high quality

Figure 7 In each landscape quality (low, medium, high), the ratio between the stabilisation time (ST) of *full* and *R-local* simulations; the ratio between the average of the execution times (AETS) of *full* and *R-local* simulations; and the ratio between the total time (TT) of *full* and *R-local* simulations; the R-LOCAL is with $R = approxR$ for $c = c_{MM}$. This is done in four different sizes of landscapes 5×50, 10×50, 15×50 and 20×50 and for different values of dispersal coefficients α.

The simulations demonstrate how the local distance R depends on two factors: the dispersal coefficient α and the landscape quality. A small dispersal coefficient α requires a large local distance R in all types of quality, while a small R is sufficient for a large α. The difference in landscape quality does not cause a significant difference in the needed value of the local distance R. Indeed, the tool of landscape network sparsification illustrates its efficiency in computing the invasion time especially for large landscapes. Even when the size of landscape is increased, the local distance R does not grow significantly (see Figure 5). This implies a sparser landscape networks for large landscapes, and therefore the time needed to compute the invasion duration decreases substantially.

As for future work, we aim to study how to improve/spoil the invasion process, e.g., to increase/decrease the speed of invasion by modifying landscapes. (Note that in some applications decreasing the speed of invasion could be more desirable, e.g., in epidemics.) This would require the computation of the invasion duration many times to verify the effectiveness of landscape modification, and therefore our work improving the speed up of the computation would be beneficial.

Acknowledgments. The authors would like to thank Dariusz Kowalski (University of Liverpool) for fruitful discussions and comments on the content of this paper.

References

1. I. Chen, J. K. Hill, R. Ohlemüller, D. B. Roy, and C. D. Thomas. Rapid range shifts of species associated with high levels of climate warming. *Science*, 333:1024–1026, 2011.
2. G. Grimmett and D. Stirzaker. *Probability and random processes.* Oxford university press, 2001.
3. J. A. Hodgson, A. Moilanen, B. A. Wintle, and C. D. Thomas. Habitat area, quality and connectivity: striking the balance for efficient conservation. *Journal of Applied Ecology*, 48(1):148–152, 2011.
4. J. A. Hodgson, C. D. Thomas, S. Cinderby, H. Cambridge, P. Evans, and J. K. Hill. Habitat recreation strategies for promoting adaptation of species to climate change. *Conservation Letters*, 4:289–297, 2011.
5. J. A. Hodgson, C. D. Thomas, C. Dytham, J. M. J. Travis, and S. J. Cornell. The speed of range shifts in fragmented landscapes. *PLoS ONE*, 7:e47141, 2012.
6. O. Honnay, K. Verheyen, J. Butaye, H. Jacquemyn, B. Bossuyt, and M. Hermy. Possible effects of habitat fragmentation and climate change on the range of forest plant species. *Ecol Lett*, 5:525–530, 2002.
7. B. Huntley, Y. C. Collingham, S. G. Willis, and R. E. Green. Potential impacts of climatic change on European breeding birds. *PLoS ONE*, 3:e1439, 2008.
8. V. G. Kulkarni. *Modeling and analysis of stochastic systems.* CRC Press, 2016.
9. M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis.* Cambridge university press, 2005.
10. D. Morton, C. Rowland, C. Wood, L. Meek, C. Marston, G. Smith, R. Wadsworth, and I. Simpson. Final report for lcm2007-the new uk land cover map. *Countryside Survey Technical Report No 11/07*, 2011.
11. M. Newman. *Networks: An Introduction.* Oxford University Press, 2010.
12. S. J. Phillips, P. Williams, G. Midgley, and A. Archer. Optimizing dispersal corridors for the cape proteaceae using network flow. *Ecological Applications*, 18:1200–1211, 2008.
13. F. Skov and J. C. Svenning. Potential impact of climatic change on the distribution of forest herbs in europe. *Ecography*, 27:366–380, 2004.
14. The Centre of Ecology and Hydrology Information Gateway. Land cover map 2007 (1km percentage aggregate class, gb) v1.2.
15. C. D. Thomas, A. Cameron, R. E. Green, et al. Extinction risk from climate change. *Nature*, 427:145–148, 2004.
16. G. R. Walther, E. Post, P. Convey, A. Menzel, C. Parmesan, T. J. Beebee, J. M. Fromentin, O. Hoegh-Guldberg, and F. Bairlein. Ecological responses to recent climate change. *Nature*, 416(6879):389–395, 2002.
17. M. S. Warren, J. K. Hill, J. A. Thomas, et al. Rapid responses of british butterflies to opposing forces of climate and habitat change. *Nature*, 414:65–69, 2001.

Ad-Hoc Affectance-Selective Families for Layer Dissemination

Harshita Kudaravalli[1] and Miguel A. Mosteiro[2]

1 **Pace University, Computer Science Department, New York, NY, USA**
`hk21040n@pace.edu`

2 **Pace University, Computer Science Department, New York, NY, USA**
`mmosteiro@pace.edu`

Abstract

Information dissemination protocols for ad-hoc wireless networks frequently use a minimal subset of the available communication links, defining a rooted "broadcast" tree. In this work, we focus on the core challenge of disseminating from one layer to the next one of such tree. We call this problem *Layer Dissemination*. We study Layer Dissemination under a generalized model of interference, called *affectance*. The affectance model subsumes previous models, such as Radio Network and Signal to Inteference-plus-Noise Ratio. We present randomized and deterministic protocols for Layer Dissemination. These protocols are based on a combinatorial object that we call *Affectance-selective Families*. Our approach combines an engineering solution with theoretical guarantees. That is, we provide a method to characterize the network with a global measure of affectance based on measurements of interference in the specific deployment area. Then, our protocols distributedly produce an ad-hoc transmissions schedule for dissemination. In the randomized protocol only the network characterization is needed, whereas the deterministic protocol requires full knowledge of affectance. Our theoretical analysis provides guarantees on schedule length. We also present simulations of a real network-deployment area contrasting the performance of our randomized protocol, which takes into account affectance, against previous work for interference models that ignore some physical constraints. The striking improvement in performance shown by our simulations show the importance of utilizing a more physically-accurate model of interference that takes into account other effects beyond distance to transmitters.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, C.2.2 Network Protocols

Keywords and phrases Wireless Networks, Broadcast Protocols, Affectance, SINR

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.33

1 Introduction

The problem of disseminating information in ad-hoc wireless communication networks (for instance, embedded in the Internet of Things) has been studied in theory and practice. To reduce traffic, dissemination protocols often use a minimal subset of the available communication links, call it T. Given that nodes communicate through radio broadcast, nodes may still receive through other links, but to provide performance guarantees only T is assumed to be available, albeit taking into account the interference of the rest of the links.

When the dissemination task involves delivery to all nodes, T defines a tree topology (since all nodes must be reachable but the set is minimal). Either because there is a single source node (e.g. [13, 14]), or because packets are first aggregated at a single node for later dissemination (e.g. [12, 15]), the problem reduces to disseminate from a root to all other

© Harshita Kudaravalli and Miguel A. Mosteiro;
licensed under Creative Commons License CC-BY

16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 33; pp. 33:1–33:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

nodes through a *Broadcast Tree*. Moreover, as we observed in [13, 14], when packets are disseminated from layer to layer in a BFS fashion, the bottleneck for fast dissemination on broadcast trees occurs at layers with high interference. Indeed, we have shown in [13, 14] that in the long run throughput is a function of maximum layer interference, and it is independent of interference in paths where packets can be easily pipelined. This phenomenon has also been observed in other works, such as in the following fragment in [7].

> *In fact, if one has a fast way of transmitting one or more messages from one layer to the next, repeating this and using pipelining would yield a fast broadcast algorithm. Thus, the crux of the broadcast problem lies in how fast can this task be solved in bipartite graphs.*

Thus, in this work, we focus on the core challenge of dissemination through one layer of a broadcast tree. We model such layer as a bipartite graph $G = (V, W, E)$ where V (the *transmitters*) and W (the *receivers*) are sets of nodes and E is the set of links from V to W. We study the Layer Dissemination problem in G assuming that initially all the transmitters have an identical piece of information, called *message* or *packet* indistinctively. To complete the task, all the nodes in W have to receive the message.

We do not assume any underlying communication infrastructure. That is, transmitters attempt to deliver the message by radio broadcast but, if two or more nodes transmit at the same time, mutual interference may prevent reception of the message. To take into account this phenomenon, we study Layer Dissemination under a general model of interference called *affectance*. As in [13, 14] we parameterize affectance with a real value $0 \leq a(u,(v,w)) \leq 1$ that represents the affectance of each transmitter u on each link (v, w). An affectance model of interference from links on links was introduced by Kesselheim [10] in the context of link scheduling. Affectance is a general model of interference in the sense that comprises other particular models studied before (cf. [14]). Moreover, previous models do not accurately represent the physical constraints in real-world deployments. For instance, in the Radio Network model [2] interference from non-neighboring nodes is neglected, and Signal to Inteference-plus-Noise Ratio (SINR) [8, 19] is a simplified model because other constraints, such as obstacles, are not taken into account.

Layer Dissemination is closely related to the combinatorial problem of computing selective families. The notion of selective families was introduced in [3] as a generalization of the dissemination problem in the Radio Network model to a combinatorial problem. Later in [4], Clementi et al. showed how to compute selective families ad-hoc, that is, for a given input family. The results are applicable to dissemination under the Radio Network model of interference when the topology is known.

In this work, we follow-up on [3] and [4] introducing the concept of *Affectance-selective Families*. That is, we generalize the dissemination problem in bipartite graphs also to a combinatorial problem, but taking into account the specific conditions to achieve a successful transmission under our generalized model of interference. Under certain conditions, we show the existence of families of subsets of $[n]$ that are affectance-selective for a given family of subsets of $[n]$. We also present randomized and deterministic distributed protocols for Layer Dissemination based on those affectance-selective families, and we provide running time theoretical guarantees.

Our approach combines an engineering solution with theoretical guarantees. That is, we provide a method to characterize the network with a global measure of affectance based on measurements of interference in the specific deployment area. Then, our protocols distributedly produce an ad-hoc transmissions schedule for dissemination. The randomized protocol only requires knowledge of the network characterization (which could be hardwired),

whereas the deterministic protocol requires full knowledge of the affectance values and it is computationally intensive. Similar approaches have been explored in practice, e.g. Conflict Maps (CMAP) [20], where nodes probe the network to build a map of conflicting transmissions.

In order to show the impact of a more accurate model of interference, we run simulations for a real-world deployment area. We compare the performance of our randomized protocol with previous protocols designed for the Radio Network and SINR models. Our experimental results expose a striking improvement in running time. Notably, this improvement does not come from algorithmic novelty, since all three protocols rely on transmitting with some probability, but from the careful choice of this probability as a function of the interference measured experimentally.

Roadmap

In Section 2 we overview previous related work. In Section 3 we specify the details of our models and the relation between affectance-selective families and dissemination in bipartite graphs. In Section 4 we specify the results obtained highlighting the novelty of our contribution. Section 5 contains our analysis and the protocols presented, and in Section 6 we present our simulation results.

2 Related Work

The literature on information dissemination in ad-hoc radio networks is vast, including a variety of models and assumptions. A lot of this work is heuristic and/or applying optimization techniques (e.g. [6, 5]) or for other models (e.g. [16, 17]). A full overview of such literature is out of the scope of this paper. We include below an overview of the most closely related work.

Before our work in [13, 14], the *generalized affectance* model was introduced and used only in the context of one-hop communication, more specifically, to link scheduling by Kesselheim [10, 11]. He also showed how to use it for dynamic link scheduling in batches. This model was inspired by the affectance parameter introduced in the more restricted SINR setting [8]. They give a characteristic of a set of links, based on affectance, that influence the time of successful scheduling these links under the SINR model. In [13, 14], we generalized this characteristic, called the maximum average tree-layer affectance, to be applicable to multi-hop communication tasks such as broadcast, together with another characteristic, called the maximum path affectance.

Layer Dissemination is closely related to the combinatorial problem of computing selective families ad-hoc for a given family of sets. The notion of selective families was introduced in [3] and it is defined as follows. Given any set of items U, a family $\mathcal{F}$ of subsets of U is called k-selective for the set U if and only if for any $X \subseteq U$, such that $|X| \leq k$, there is a set $Y \in \mathcal{F}$ satisfying $|X \cap Y| = 1$. Here, we introduce the concept of affectance-selective families, taking into account the specific conditions to achieve a successful transmission under affectance.

With respect to selective families, our work can be seen as an extension of [4] to affectance. Indeed, in [4], Clementi et al. showed how to compute selective families ad-hoc for a given family. That is, their algorithm can be used for dissemination under the Radio Network model. For instance, the input families can be seen as the different subsets of nodes that may be active at a given time, or as in Layer Dissemination as the subsets of transmitters connected to each receiver. Here, we revisit this problem under affectance, that is, we show the existence of affectance-selective families (the precise notion is defined in Section 3), we

present randomized and deterministic protocols to solve Layer Dissemination based on the affectance-selective families, and we analyze their performance.

3 Model and Problem

We model the network topology as a bipartite graph $G = (V, W, E)$, where V is the set of transmitters, W is the set of receivers, $|V| = |W| = n$, and E is the set of links from V to W. That is, for every $(v, w) \in E$, we have $v \in V$ and $w \in W$. For each $w \in W$, we denote by E_w the set of links incoming to receiver w, and by F_w the set of transmitters of those links.

Following [14], we model the interference among transmissions with an ***affectance matrix***

$$A = \Big[a(u, (v, w))\Big]_{\substack{u \in V \\ (v,w) \in E}} ,$$

where $a(u, (v, w))$ is a real number in $[0, 1]$ quantifying the interference that the transmitter u introduces to the communication through link (v, w). We denote $a_{V'}((v, w))$ as the total affectance of a set of transmitters $V' \subseteq V$ on a link (v, w) (i.e., the sum of affectances on (v, w) over all nodes in V'), and further, $a_{V'}(E')$ as the total affectance of a set of transmitters $V' \subseteq V$ on a set of links $E' \subseteq E$ (i.e., the sum of affectances of V' over all links in E'). We do not restrict the affectance function, as long as its effect is additive; that is,

$$a_{V'}((v, w)) = \sum_{u \in V'} a(u, (v, w)) \text{ , and}$$

$$a_{V'}(E') = \sum_{(v,w) \in E'} a_{V'}((v, w)) \ .$$

Without loss of generality we assume that time is slotted. Then, under the above affectance model, a ***successful transmission*** in a time slot t is defined as follows. For any link $(v, w) \in E$, a transmission from v is received at w in time slot t if and only if:

- v transmits in time slot t, and
- $a_{\mathcal{T}(t)}((v, w)) < 1$, where $\mathcal{T}(t) \subseteq V$ is the set of nodes transmitting in time slot t.

The event of a non-successful transmission, that is, when the affectance is at least 1, is called a ***collision***. We assume that a node listening to the channel cannot distinguish between a collision and background noise present in the channel in absence of transmissions; in other words, the model is without collision detection.

Under the model above, the ***Layer Dissemination*** problem is defined as follows: for each node $w \in W$, w must receive a successful transmission from some node in F_w.

We define affectance-selective families as a purely combinatorial problem on a family of subsets of integers and a matrix of real numbers. (Refer to Section 2 for a definition of classic selective families.) The relation with Layer Dissemination is the following. For each receiver $w \in W$, consider the set $F_w \subseteq V$ of transmitters connected to w. These sets of transmitters define a family $\mathcal{F}$ of subsets of nodes in V. On the other hand, for a given Layer Dissemination protocol, the schedule of transmissions from nodes in V can also be viewed as a family $\mathcal{S}$ of subsets of nodes. Specifically, for each time slot t, the subset of nodes in V transmitting in t is a member of $\mathcal{S}$. In the Radio Network model, the family $\mathcal{S}$ is called selective on the family $\mathcal{F}$ if and only if for any $F_w \in \mathcal{F}$ there is some $S_t \in \mathcal{S}$ such that $|S_t \cap F_w| = 1$. This is because w successfully receives a message if and only if exactly one node in F_w transmits. Given an integer $n > 0$, consider a family $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ of subsets of integers in $[n]$. Let A be a matrix of real numbers in $[0, 1]$ defined on $\mathcal{F}$ in such

a way that for each $u \in [n]$ there is a corresponding row, and for each $w \in [n]$ and each $v \in F_w$ there is a column in A corresponding to the pair (v, w). Then, we say that a family $\mathcal{S} = \{S_1, S_2, \ldots, S_s\}$ of subsets of $[n]$ is ***affectance-selective*** on the family $\mathcal{F}$ if for each $w \in [n]$ there exists $j \in [s]$ such that:

- $|F_w \cap S_j| \geq 1$, and
- for some $v \in (F_w \cap S_j)$ it is $\sum_{u \in S_j} a(u, (v, w)) < 1$.

We say that the family $\mathcal{S}$ has ***length*** s, and that each w is ***affectance-selected***, or simply ***selected*** for short.

In terms of Layer Dissemination, labeling the transmitters as well as the receivers with consecutive integers in $[n]$, each $F_w \in \mathcal{F}$ is the subset of transmitters connected to receiver w, A is the affectance matrix, and each value $a(u, (v, w))$ in A corresponds to the affectance of node u on link (v, w). Then, the family $\mathcal{S}$ is a solution for Layer Dissemination setting each node in set $S_t \in \mathcal{S}$ to transmit in time slot t, for each $t \in [s]$.

4 Our Results

In this work, for a given family $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ of subsets of integers in $[n]$ and a given affectance matrix A, we first show the existence of a family $\mathcal{S}$ of subsets of $[n]$ that is affectance-selective on $\mathcal{F}$. Under certain conditions on the relation between $\mathcal{F}$ and A, the family $\mathcal{S}$ is proved to have a number of sets that is in $O(1 + \log n \log \overline{A})$. That is, at most logarithmic on n and logarithmic on the ***maximum average affectance*** $\overline{A}$. The latter is a characterization based on $\mathcal{F}$ and A. Specifically,

$$\overline{A} = \max_{w \in [n]} \max_{F \subseteq F_w} \sum_{v \in F} \sum_{u \in [n]} a(u, (v, w))/|F|.$$

The condition assumed is that the maximum average affectance is not more than an a constant factor larger than the maximum degree. This is a fair assumption for multi-hop radio networks given that interference is a local restriction rather than local.

The proof of that bound is existential because it is based on the probabilistic method (as in [4]). Nevertheless, it provides a method to derive algorithms for Layer Dissemination. We present two Layer Dissemination distributed protocols, one randomized and one deterministic. We show that both protocols have the same running time guarantee, which is asymptotically the same as the size of the affectance-selective family shown. That is, $O(1 + \log n \log \overline{A})$. The randomized protocol is Monte Carlo, it is very simple (a version of Decay [1]), and only requires knowledge of n, $\overline{A}$, and two constants. The deterministic protocol (inspired on [4]) provides worst-case guarantees, but nodes need to know the topology and the affectance matrix A, and its computational complexity is exponential.

We also include simulations to evaluate the impact of using a more accurate model of interference. We compare our randomized protocol with previous work for the Radio Network and SINR models. Our experimental results show a striking improvement in performance because the Radio Network protocol neglects interference from non-neighboring nodes, whereas SINR protocols do not take advantage of low interference from nodes that, although located at a short distance, are blocked by obstacles. Our results also show that for the particular inputs tested our randomized protocol performs better than predicted by our theoretical analysis.

Algorithm 1: Randomized Layer Dissemination protocol for each node $v \in V$. $\overline{A} = \max_{w\in W} \overline{A}_w$, is the maximum average affectance, where $\overline{A}_w = \max_{F\subseteq F_w} \sum_{v\in F} \sum_{u\in V} a(u,(v,w))/|F|$ is the maximum average affectance on w, $d < 1$ is a constant as computed in the proof of Theorem 1, and $c > 1$ is the constant bounding $\overline{A}_w \leq c|F_w|$ for each receiver w.

```
1 b ← 1 + 1/(2c)
2 m ← ⌈2 log_{1/d} n⌉
3 for i = 0, 1, 2, ..., max{⌈log_b(2Ā)⌉, 0} do
4 |  for m times do
5 |  |  transmit with probability 1/b^i
```

5 Analysis

5.1 Existence of an Affectance-selective Family of Polylogarithmic Size

The proof of the following theorem, based on the probabilistic method, is left to the Appendix.

▶ **Theorem 1.** *For any $n > 0$, consider a family $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$ of subsets of integers in $[n]$ and any affectance matrix A defined on $\mathcal{F}$. For each $w \in [n]$, let $\overline{A}_w = \max_{F\subseteq F_w} \sum_{v\in F} \sum_{u\in[n]} a(u,(v,w))/|F|$ be the maximum average affectance on w. If there exists a constant $c > 1$ such that $\overline{A}_w \leq c|F_w|$ for all $w \in [n]$, then, there exists a family $\mathcal{S} = \{S_1, S_2, \dots, S_s\}$ that is affectance-selective on $\mathcal{F}$, and its size s satisfies*

$$s \in O\left(1 + \log n \log \overline{A}\right),$$

where $\overline{A} = \max_{w\in[n]} \overline{A}_w$ is the maximum average affectance.

The bound shown matches the $O(1 + \log\Delta \log|\mathcal{F}|)$ bound for the Radio Network model in [4], because in our setting the number of subsets to select is $|\mathcal{F}| = n$, and in the Radio Network model it is $\overline{A} = \Delta - 1$.

5.2 Randomized Layer Dissemination Protocol

The proof of Theorem 1, showing the existence of an affectance-selective family, yields a Monte Carlo distributed randomized protocol for Layer Dissemination applicable to settings where the conditions of the theorem hold. I.e., there exists a constant c bounding $\overline{A}_w \leq c|F_w|$ for each receiver w. The protocol requires that all transmitters have knowledge of the maximum average affectance $\overline{A}$, the constant c, the number of transmitters n, and the constant $d < 1$ computed in the proof of Theorem 1. The protocol, detailed in Algorithm 1, is a version of the Decay protocol [1] extended to the affectance model. Its correctness and running time are established in the following theorem.

▶ **Theorem 2.** *Consider a layer of a Radio Network with affectance matrix A and topology $G = (V, W, E)$, where $|V| = |W| = n$, where for each receiver $w \in W$ there is at least one transmitter $v \in V$ such that $(v, w) \in E$. Then, if there exists a constant $c > 1$ such that $\overline{A}_w \leq c|F_w|$ for all $w \in W$, where $\overline{A}_w = \max_{F\subseteq F_w} \sum_{v\in F} \sum_{u\in V} a(u,(v,w))/|F|$ is the maximum average affectance on w, Algorithm 1 solves the Layer Dissemination problem with high probability [1], and the running time is in $O(1 + \log n \log \overline{A})$, where $\overline{A} = \max_{w\in W} \overline{A}_w$ is the maximum average affectance.*

[1] We say that an event occurs *with high probability* if it occurs with probability at least $1 - 1/n^{\kappa}$, for some constant $\kappa > 0$.

Proof. The first claim follows from the proof of Theorem 1, together with computing the value m that makes $Pr(\exists w \in [n] : Z_w = 0) \leq nd^m \leq 1/n$. The running time follows from the number of iterations in Algorithm 1. ◀

For settings where only n and c are known to the transmitters, we can run the loop in Line 3 of Algorithm 1 for $\lceil \log_b(2(n-1)) \rceil$ times, since we know that $\overline{A}_w \leq (n-1)$ for any $w \in W$. The running time in that case would be $1 + O(\log^2 n)$ steps.

5.3 Deterministic Layer Dissemination Protocol

Algorithm 1 is simple and it is easily distributed because only requires knowledge of a few global parameters (namely $\overline{A}$, c, and n), and also does not require intensive computations at each node. However, the running time guarantee is only stochastic. In this section we present a deterministic algorithm that provides the same running time guarantee but worst-case, although to implement it distributedly knowledge of the graph G and the affectance matrix A is required.

The ideas of algorithm $greedy_{MSF(\Delta)}$ [4] can be re-used here to compute a transmission schedule that solves Layer Dissemination, but $greedy_{MSF(\Delta)}$ cannot be used as-is because it does not cope with affectance or families of sets with different sizes. So, building upon the ideas of $greedy_{MSF(\Delta)}$, we present in this section an algorithm for Layer Dissemination under the affectance model. That is, the transmission schedule is computed to cope with affectance, and without assuming anything about the number of neighbors of each receiver. We specify such protocol in Algorithm 2 and an explanation of the details follow.

The receivers pending to be selected (initially all) are partitioned in subsets so that, for each receiver w, it is

$$w \in \begin{cases} W'_0 & \text{if } \overline{A}_w \leq 1/(2b) \\ W'_r & \text{if } b^{r-1}/2 < \overline{A}_w \leq b^r/2, \text{ for } r = 0, 1, \ldots, m. \end{cases}$$

The expectations in Lines 12 and 13 of the protocol correspond to the following. Recall that we assume the transmitters to be labeled by consecutive integers. That is, the set of transmitters is $V = \{1, 2, \ldots, n\}$. Then, in Algorithm 2, for each time slot t, we keep track of whether each node in V transmits or not in an array of booleans V', where index i of the array is true if i transmits in t and false otherwise. The array is filled incrementally for $i = 1, 2, \ldots, n$ as follows. For each index i, let $V_{>i} = \{i+1, \ldots, n\}$ if $i < n$, or $V_{>i} = \emptyset$ otherwise. Likewise, let $V_{<i} = \{1, \ldots, i-1\}$ if $i > 1$, or $V_{<i} = \emptyset$ otherwise.

Then, for each value of $r = 0, 1, \ldots$, taking into account the action of transmitters in $V_{<i}$ that was already decided, we decide whether transmitter i transmits or not in t computing the expected number of receivers from a given subset that will be affectance-selected, if i transmits and the actions of transmitters in $V_{>i}$ is chosen at random with probability b^{-r} (Line 12). We do the same for the case that transmitter i does not transmit (Line 13). The expectations are taken over the random choice of transmitters in $V_{>i}$. Such computation is feasible given that every transmitter $v \in V$ is assumed to know $G = (V, W, E)$ and the affectance matrix A. The specific computation of expectations is the following.

The calculation corresponds to the ith iteration of the inner loop (Line 11) and probability $p = b^{-r}$ for some r. Let $X_{v,i}$ be an indicator variable defined as follows. The variable $X_{v,i}$ is random if $v \in V_{>i}$, and deterministic otherwise. For each $v \in V_{<i}$, $X_{v,i} = 1$ if and only if $V'[v] = true$. For each $v \in V_{>i}$, $X_{v,i} = 1$ with probability p or $X_{v,i} = 0$ with probability $1 - p$. Finally, it is $X_{i,i} = 1$ to compute the expectation $\mathbb{E}_{true}$ (Line 12) or $X_{i,i} = 0$ to

Algorithm 2: Deterministic Layer Dissemination protocol for each node $v \in V$. $\overline{A} = \max_{w \in W} \overline{A}_w$, is the maximum average affectance, where $\overline{A}_w = \max_{F \subseteq F_w} \sum_{v \in F} \sum_{u \in V} a(u,(v,w))/|F|$ is the maximum average affectance on w, and $c > 1$ is the constant bounding $\overline{A}_w \leq c|F_w|$ for each w.

```
    // Initialization
 1  p ← 0
 2  b ← 1 + 1/(2c)
 3  m ← max{⌈log_b(2Ā)⌉, 0}
 4  W'_0 ← {w ∈ W : Ā_w ≤ 1/2}
 5  for r = 1,...,m do  W'_r ← {w ∈ W : b^{r−1}/2 < Ā_w ≤ b^r/2}
    // Protocol
 6  for each time slot while ∃r = 0,1,...,m : W'_r ≠ ∅ do
 7  |  if p ≤ 1/(2bĀ) then
 8  |  |  p ← 1
 9  |  |  r ← 0
10  |  set V'[1...n] array of booleans                    // V'[i] ≡ i transmits
11  |  for i = 1,2,...,n do
12  |  |  E_true ← E_{V'[i+1...n]} (# selected in W'_r | V'[i] = true)
13  |  |  E_false ← E_{V'[i+1...n]} (# selected in W'_r | V'[i] = false)
14  |  |  V'[i] ← E_true > E_false
15  |  if V'[v] then transmit
16  |  W'_r ← W'_r \ {w | w was selected}
17  |  p ← p/b
18  |  r ← r + 1
```

compute the expectation $\mathbb{E}_{false}$ (Line 13). Also, let $Z_{w,i}$ be a random variable indicating whether receiver w is selected or not.

Then, it is

$$\mathbb{E}_{V'[i+1\ldots n]}\left(\#\text{ selected in } W'_r \middle| V'[i] = true\right) = \sum_{w \in W'_r} Z_{w,i} Pr(Z_{w,i} = 1 | X_{i,i} = 1)$$
$$\mathbb{E}_{V'[i+1\ldots n]}\left(\#\text{ selected in } W'_r \middle| V'[i] = false\right) = \sum_{w \in W'_r} Z_{w,i} Pr(Z_{w,i} = 1 | X_{i,i} = 0).$$

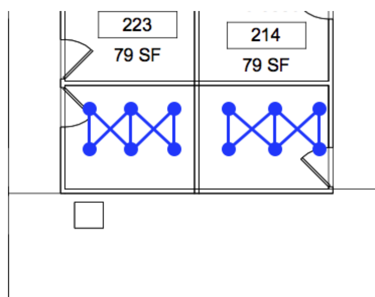
Where,

$$Pr(Z_{w,i} = 1) = Pr\left(\sum_{v \in F_w} X_{v,i} \geq 1 \text{ and } \exists v \in F_w : \sum_{u \in V} \sum_{v \in F_w} a(u,(v,w)) X_{u,i} X_{v,i} < 1\right).$$

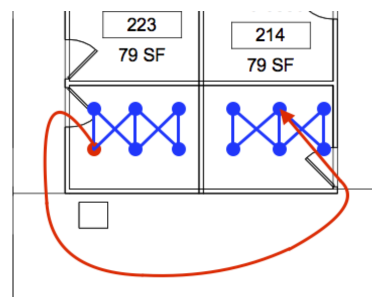
In the following theorem, we prove that each time the probability p is updated to 1 (Line 8), at least a constant fraction of receivers is selected, solving Layer Dissemination in a logarithmic number of steps. The proof is left to the appendix.

▶ **Theorem 3.** *Consider a layer of a Radio Network with affectance matrix A and topology $G = (V, W, E)$, where $|V| = |W| = n$, where for each receiver $w \in W$ there is at least one transmitter $v \in V$ such that $(v, w) \in E$. Then, if there exists a constant $c > 1$ such that $\overline{A}_w \leq c|F_w|$ for all $w \in W$, where $\overline{A}_w = \max_{F \subseteq F_w} \sum_{v \in F} \sum_{u \in V} a(u,(v,w))/|F|$ is the maximum average affectance on w, Algorithm 2 solves the Layer Dissemination problem, and the running time is in $O(1 + \log n \log \overline{A})$, where $\overline{A} = \max_{w \in W} \overline{A}_w$ is the maximum average affectance.*

(a) Seidenberg School of CSIS floor plan. **(b)** A layer of the network grid.



(c) The network topology.



(d) Example of affectance.

Figure 1 Illustration of network deployment.

6 Simulations

In this section we present our simulations, developed to evaluate the impact of a more accurate model of interference on Layer Dissemination. For that purpose, we run simulations for a real-world deployment area, comparing the performance of our randomized protocol with previous protocols designed for the Radio Network and SINR models. The details follow.

We used as a model of a network deployment area the floor plan of the Seidenberg School of Computer Science and Information Systems at Pace University, considering nodes installed in the intersections of each square of four ceiling panels (see Figure 1a). To evaluate Layer Dissemination, we focused on one layer of this network going across various offices (see Figure 1b). For simplicity, to evaluate performance as n grows, we replicated the same office multiple times in a layer.

The walls of these offices have a metallic structure. Hence, each office behaves as a Faraday cage blocking radio transmissions (specially millimeter wave). Consequently, most of the radio waves propagate through doors (which are not metallic). We fixed the radio transmission power to be large enough to reach five grid cells, so that transmissions from layer to layer are possible. So, given the offices dimensions, transmitters within an office are connected to all receivers. On the other hand, the interference to other offices in the same layer is approximated by adding ten grid cells for each office of distance. The resulting

Algorithm 3: Decay protocol [1] for each transmitter $v \in V$. Δ is the maximum in-degree of the network.

```
1  rounds ← 0
2  counter ← 0
3  while ∃w ∈ W : w did not receive do
4  |  rounds + +
5  |  if counter = 0 then transmit ← true
6  |  if transmit = true then
7  |  |  v transmits the message
8  |  |  with probability 1/2 set transmit ← false
9  |  counter + +
10 |  if counter = 2⌈log Δ⌉ then counter ← 0
11 return rounds
```

topology can be seen in Figure 1c, whereas the reason why affectance is more accurate than interference based on Euclidean distance is illustrated in Figure 1d. For instance, it can be seen that transmitters that are close to a wall in one office have low affectance on links that are close to other side of that wall in the contiguous office, even though they are separated by only one grid-cell in Euclidean distance.

Using the network topology and the resulting affectance matrix described above as input, and for $n = 6, 9, 12, \ldots, 42$, we simulated our randomized protocol in Algorithm 1, which requires knowledge of only global variables n, c, and $\overline{A}$. (Refer to Algorithm 1 for further details.) For comparison, we also simulated protocols designed for the Radio Network and SINR models of interference on the same inputs, but considering a transmission successful under the affectance model constraints, as defined in Section 3. We did not simulate our deterministic protocol in Algorithm 2 because the schedule computation has exponential complexity.

For the Radio Network model of interference, we simulated the classic Decay [1] protocol, whereas for SINR we simulated the Broadcast protocol in Algorithm 1 in [9]. (Most of the work for SINR is oriented to link scheduling, which cannot be accurately mapped to Layer Dissemination or Broadcast.) The former requires knowledge of global variable Δ, which is the maximum in-degree in the network, whereas the latter requires knowledge of global variables *density* and *dilution*, as defined in [9]. All three protocols provide guarantees on the number of rounds of communication needed to complete Broadcast, but running them for that fixed time would not provide any performance comparison. Instead, for each of the protocols we measured the number of rounds of communication passed until all receivers have received the message. In Algorithms 3 and 4 we specify how we adapted the Radio Network and SINR protocols respectively for our simulations. The Java code of our simulator can be found at `http://csis.pace.edu/~mmosteiro/pub/sourceLayerDiss/`. The results of the simulations are plotted in Figure 2 and analyzed in the following section.

7 Discussion

As seen in the plot of Figure 2, our experimental results show a striking improvement in performance of our protocol with respect to Algorithms 3 and 4. Indeed, the running times of Algorithms 3 and 4 grow exponentially with n (the scale of the y axis is logarithmic), whereas our algorithm's running time grows exponentiallly slower. Moreover, the plot shows also the

Algorithm 4: Algorithm 1 in [9] for each transmitter $v \in V$. *density* and *dilution* are parameters of the network as defined in [9].

1 $rounds \leftarrow 0$
2 **while** $\exists w \in W : w$ *did not receive* **do**
3 $\quad rounds++$
4 $\quad$ **if** $rounds \equiv v \mod dilution$ **then**
5 $\quad\quad$ with probability $1/density$, v transmits the message
6 **return** $rounds$

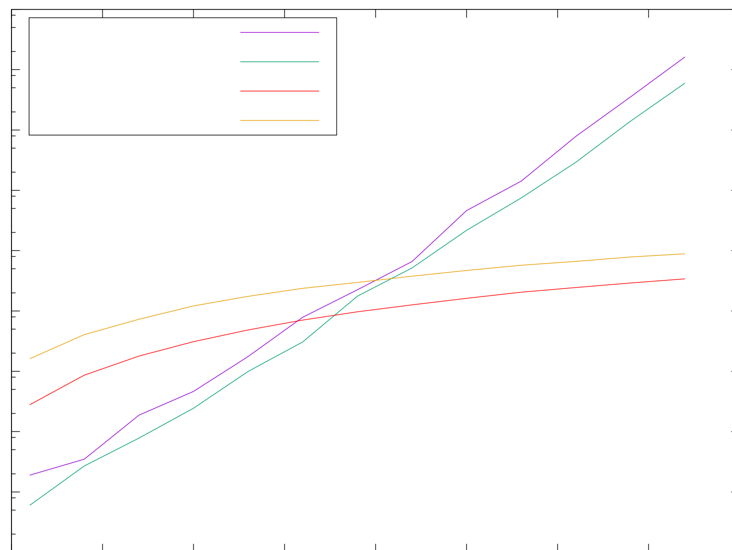


Figure 2 Simulation results.

theoretical upper bound proved in Theorem 2. It can be seen that in these simulations our protocol performs better than the theoretical guarantees. This difference in performance could be due to an algorithmic improvement. However, at their core, all three algorithms are based on iteratively choosing to transmit with some probability. Thus, we conclude that the improvement is due to a careful choice of such transmission probability, making it a function of the network characteristic derived from the interference measured experimentally, rather than due to algorithmic novelty. This conclusion should not come as a surprise, given that Algorithm 3 was designed neglecting interference from non-neighboring nodes, whereas Algorithm 4 does not take advantage of low interference from nodes that, although located at a short distance, are blocked by obstacles. Therefore, the results of our experimental evaluation show the importance of studying information dissemination under more accurate models of interference.

Acknowledgements. The authors want to thank Dariusz R. Kowalski for seminal ideas and thorough discussions that led to the development of this work.

References

1. Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.
2. Imrich Chlamtac and Shay Kutten. Tree-based broadcasting in multihop radio networks. *IEEE Trans. Computers*, 36(10):1209–1223, 1987.
3. Bogdan S. Chlebus, Leszek Gasieniec, Alan Gibbons, Andrzej Pelc, and Wojciech Rytter. Deterministic broadcasting in ad hoc radio networks. *Distributed Computing*, 15(1):27–38, 2002. doi:10.1007/s446-002-8028-1.
4. Andrea EF Clementi, Pilu Crescenzi, Angelo Monti, Paolo Penna, and Riccardo Silvestri. On computing ad-hoc selective families. In *Proc. of the 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and 5th International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 2129 of *Lecture Notes in Computer Science*, pages 211–222, 2001.
5. Fabio D'Andreagiovanni, Carlo Mannino, and Antonio Sassano. GUB covers and power-indexed formulations for wireless network design. *Management Science*, 59(1):142–156, 2013. doi:10.1287/mnsc.1120.1571.
6. Peter Dely, Fabio D'Andreagiovanni, and Andreas Kassler. Fair optimization of mesh-connected WLAN hotspots. *Wireless Communications and Mobile Computing*, 15(5):924–946, 2015. doi:10.1002/wcm.2393.
7. Mohsen Ghaffari, Bernhard Haeupler, and Majid Khabbazian. The complexity of multi-message broadcast in radio networks with known topology. *CoRR*, abs/1205.7014, 2012.
8. Magnús M. Halldórsson and Roger Wattenhofer. Wireless communication is in apx. In *Proc. of the 36th International Colloquium on Automata, Languages and Programming, Part I*, pages 525–536, 2009.
9. Tomasz Jurdzinski, Dariusz R. Kowalski, Michal Rozanski, and Grzegorz Stachowiak. Distributed randomized broadcasting in wireless networks under the sinr model. In Yehuda Afek, editor, *DISC*, volume 8205 of *Lecture Notes in Computer Science*, pages 373–387. Springer, 2013.
10. Thomas Kesselheim. Dynamic packet scheduling in wireless networks. In *Proc. of the 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 281–290, 2012.
11. Thomas Kesselheim and Berthold Vöcking. Distributed contention resolution in wireless networks. In *Proc. of the 24th International Symposium on Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 163–178. Springer-Verlag, Berlin, 2010.
12. Majid Khabbazian and Dariusz R. Kowalski. Time-efficient randomized multiple-message broadcast in radio networks. In Cyril Gavoille and Pierre Fraigniaud, editors, *PODC*, pages 373–380. ACM, 2011.
13. Dariusz R. Kowalski, Miguel A. Mosteiro, and Tevin Rouse. Dynamic multiple-message broadcast: bounding throughput in the affectance model. In *10th ACM International Workshop on Foundations of Mobile Computing, FOMC 2014, Philadelphia, PA, USA, August 11, 2014*, pages 39–46, 2014.
14. Dariusz R. Kowalski, Miguel A. Mosteiro, and Kevin Zaki. Dynamic multiple-message broadcast: Bounding throughput in the affectance model. *CoRR*, abs/1512.00540, 2015. URL: http://arxiv.org/abs/1512.00540.
15. Fredrik Manne and Qin Xin. Optimal gossiping with unit size messages in known topology radio networks. In *Workshop on Combinatorial and Algorithmic Aspects of Networking*, pages 125–134. Springer, 2006.
16. Gianluca De Marco and Dariusz R. Kowalski. Towards power-sensitive communication on a multiple-access channel. In *2010 International Conference on Distributed Computing*

Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010, pages 728–735, 2010. doi:10.1109/ICDCS.2010.50.

17 Gianluca De Marco and Dariusz R. Kowalski. Fast nonadaptive deterministic algorithm for conflict resolution in a dynamic multiple-access channel. *SIAM J. Comput.*, 44(3):868–888, 2015. doi:10.1137/140982763.

18 Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge University Press, 2005.

19 Christian Scheideler, Andréa W. Richa, and Paolo Santi. An o(log n) dominating set protocol for wireless ad-hoc networks under the physical interference model. In *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 91–100. ACM, 2008.

20 Mythili Vutukuru, Kyle Jamieson, and Hari Balakrishnan. Harnessing exposed terminals in wireless networks. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 59–72, 2008.

A Appendix

A.1 Proof of Theorem 1

Proof. We prove the claim using the probabilistic method. That is, we show a randomly generated family $\mathcal{S}$ such that the probability that $\mathcal{S}$ does not select some set in $\mathcal{F}$ is strictly less than one.

Let S be a subset of $[n]$ defined as follows. For each $v \in [n]$, independently include v in S with some probability p (we will discuss the best choice for p later). Let X_v be a random variable indicating whether v is in S or not. Let Z_w be a random variable indicating whether $w \in [n]$ is selected or not. The probability that w is not selected given that some $v \in F_w$ is in S is

$$Pr\left(Z_w = 0 \middle| \sum_{v \in F_w} X_v \geq 1\right) \leq Pr\left(\sum_{u \in [n]} \sum_{v \in F_w} a(u,(v,w)) X_u X_v \geq \sum_{v \in F_w} X_v\right).$$

The above inequality is true because, for w not to be selected, the affectance in all pairs (v, w) such that $v \in F_w$ and $X_v = 1$ has to be at least one. The inequality is due to the right-hand side possibly including events where some pairs have affectance less than one, but others have affectance larger than one so that the overall sum is still larger than $\sum_{v \in F_w} X_v$. This right-hand side can be bounded as follows using a Markov-type inequality that can be proved as in [18].

$$Pr\left(\sum_{u \in [n]} \sum_{v \in F_w} a(u,(v,w)) X_u X_v \geq \sum_{v \in F_w} X_v\right) \leq E\left(\frac{\sum_{u \in [n]} \sum_{v \in F_w} a(u,(v,w)) X_u X_v}{\sum_{v \in F_w} X_v}\right).$$

Replacing

$$\overline{A}_w = \max_{F \subseteq F_w} \sum_{v \in F} \sum_{u \in [n]} a(u,(v,w))/|F| \geq \sum_{v \in F'} \sum_{u \in [n]} a(u,(v,w))/|F'|,$$

for any $F' \subseteq F_w$, we get the following bound.

$$Pr\left(Z_w = 0 \middle| \sum_{v\in F_w} X_v \geq 1\right) \leq E\left(\frac{\sum_{u\in[n]}\sum_{v\in F_w} a(u,(v,w))X_u X_v}{\sum_{v\in F_w} X_v}\right)$$
$$= E\left(\sum_{u\in[n]} \frac{\sum_{v\in F_w} a(u,(v,w))X_v}{\sum_{z\in F_w} X_z} X_u\right)$$
$$\leq \overline{A}_w p\ .$$

Then, we have that

$$\begin{aligned} Pr\left(Z_w = 0\right) &= Pr\left(Z_w = 0 \middle| \sum_{i\in F_w} X_i \geq 1\right) Pr\left(\sum_{i\in F_w} X_i \geq 1\right) \\ &+ Pr\left(Z_w = 0 \middle| \sum_{i\in F_w} X_i = 0\right) Pr\left(\sum_{i\in F_w} X_i = 0\right) \\ &= Pr\left(Z_w = 0 \middle| \sum_{i\in F_w} X_i \geq 1\right) (1-(1-p)^{|F_w|}) + (1-p)^{|F_w|} \\ &\leq \overline{A}_w p(1-(1-p)^{|F_w|}) + (1-p)^{|F_w|} \qquad (1) \\ &= \overline{A}_w p + (1-\overline{A}_w p)(1-p)^{|F_w|}. \end{aligned}$$

Consider now a family $\mathcal{S} = \{S_i\}$ of subsets of $[n]$ where S_i is obtained including each $v \in [n]$ independently with probability $p = 1/b^i$ for $i = 0, 1, 2, \ldots, \max\{\lceil \log_b(2\overline{A})\rceil, 0\}$ and $b = 1 + 1/(2c)$. If $\overline{A}_w \leq 1/(2b)$, replacing in Equation 1 we have that $Pr\left(Z_w = 0\right) \leq 1/(2b)$ for $p = 1$, which is strictly smaller than 1. Otherwise, if $\overline{A}_w > 1/(2b)$, we know that, for some i, it is $1/(2b\overline{A}_w) < p \leq 1/(2\overline{A}_w)$. Replacing,

$$Pr\left(Z_w = 0\right) \leq \frac{1}{2} + \left(1 - \frac{1}{2b}\right)\left(1 - \frac{1}{2b\overline{A}_w}\right)^{|F_w|}.$$

Using that $\overline{A}_w \leq c|F_w|$ for some constant $c > 1$, we obtain

$$\begin{aligned} Pr\left(Z_w = 0\right) &\leq \frac{1}{2} + \left(1 - \frac{1}{2b}\right)\left(1 - \frac{1}{2bc|F_w|}\right)^{|F_w|}, \text{ using that } 2bc|F_w| > 1, \\ &\leq \frac{1}{2} + \left(1 - \frac{1}{2b}\right)\left(\frac{1}{e}\right)^{1/(2bc)}. \end{aligned}$$

Replacing $c = 1/(2(b-1))$ we get

$$Pr\left(Z_w = 0\right) \leq \frac{1}{2} + \left(1 - \frac{1}{2b}\right)\left(\frac{1}{e}\right)^{(b-1)/b}.$$

To show that there is a positive probability that w is selected, we show that for each constant

c there is a constant $b = 1 + 1/(2c)$ such that the latter is strictly smaller than 1 as follows.

$$\frac{1}{2} + \left(1 - \frac{1}{2b}\right)\left(\frac{1}{e}\right)^{(b-1)/b} < 1$$
$$\left(1 - \frac{1}{2b}\right)\left(\frac{1}{e}\right)^{(b-1)/b} < \frac{1}{2}$$
$$1 - \frac{1}{2b} < \frac{1}{2}e^{(b-1)/b}$$
$$1 - \frac{1}{2}e^{(b-1)/b} < \frac{1}{2b}$$
$$2b - be^{(b-1)/b} < 1.$$

The left hand side is equal to 1 for $b = 1$ and monotonically decreasing for any b such that $1 < b < 1.5$, which is the range of $b = 1 + 1/(2c)$ for any $c > 1$.

Having proved that that there is a positive probability that w is selected, we add a multiplicity m on the sets S_i to show that the probability that *some* $w \in [n]$ is not selected is small, as follows.

We redefine $\mathcal{S}$ as the family $\{S_{i,j}\}$ of subsets of $[n]$ where the set $S_{i,j}$ is obtained including each $v \in [n]$ in $S_{i,j}$ independently with probability $p = 1/b^i$, for each $i = 0, 1, 2, \ldots, \max\{\lceil\log_b(2\overline{A})\rceil, 0\}$ and each $j = 1, 2, \ldots, m$.

Then, the probability that a given w is not selected is $Pr(Z_w = 0) \leq d^m$, where $d < 1$ is some constant as shown above. Using the union bound, the probability that *some* $w \in [n]$ is not selected is $Pr(\exists w \in [n] : Z_w = 0) \leq nd^m$, which is smaller than 1 for some $m \in \Theta(\log n)$, showing the existence of an affectance-selective family $\mathcal{S}$ of size $O(1 + \log n \log \overline{A})$. ◀

A.2 Proof of Theorem 3

Proof. Algorithm 2 is correct as long as it terminates, as it does not stop until $W' = \emptyset$ (Line 6). Then, to prove the claim, it is enough to prove the upper bound on the running time, which we do as follows.

Consider the execution divided in stages, where a new stage starts each time that p is set to 1 (Line 1 and Line 8). Moreover, consider each stage divided in rounds according to the value of r. That is, starting from round $r = 0$ when $p = 1$, a new round starts each time that p and r are updated in Lines 17 and 18. Thus, each stage is composed by rounds $0, 1, 2, \ldots, m$ when $p = 1, b^{-1}, b^{-2}, \ldots, b^{-m}$ respectively, and when p becomes smaller or equal than $1/(2b\overline{A})$, a new stage begins and p is reset to 1 in Line 8.

We show now that, in any given round r, a constant fraction of receivers in W'_r is selected. Thus, a constant fraction of receivers is selected in each stage, which yields $O(\log n)$ stages, each of $O(\log \overline{A})$ rounds, proving the claimed running time.

Fix any given round r when $p = b^{-r}$. We focus then on showing that a constant fraction of receivers in W'_r is selected, knowing that, for each receiver $w \in W'_r$, if $r = 0$ it is $\overline{A}_w \leq 1/2$, and if $r > 0$ it is $b^{r-1}/2 < \overline{A}_w \leq b^r/2$.

We showed in the proof of Theorem 1 that, for any $w \in [n]$, if a subset $S \subseteq [n]$ is chosen including each $v \in [n]$ with a probability b^{-i}, for i such that $1/(2b\overline{A}_w) < b^{-i} \leq 1/(2\overline{A}_w)$, the probability of selecting w with S is a positive constant q. The specific bound on q is dependent on whether $\overline{A}_w \leq 1/(2b)$ or not, but still a constant for both cases. This bound applies to round r for any receiver $w \in W'_r$ and S a subset of transmitters, each chosen with probability b^{-r}. Thus, the expected number of receivers selected by S from W'_r would be qW'_r,

that is, a constant fraction q. Let this expectation be denoted as $\mathbb{E}_{X[1\ldots n]}(\#\text{ selected in } W'_r)$, where each $X[i]$ indicates whether $i \in S$.

Then, to complete the proof, now we show that the expected number of receivers selected from W'_r by the set of transmitters defined by the array V' after completing the loop in Lines 11-14 (which indeed is the actual number because no random choice is made in the last iteration) is at least $\mathbb{E}_{X[1\ldots n]}(\#\text{ selected in } W'_r)$. Indeed, we prove the stronger claim that $\max\{\mathbb{E}_{true}, \mathbb{E}_{false}\} \geq \mathbb{E}_{X[1\ldots n]}(\#\text{ selected in } W'_r)$ for each iteration of the loop, which we show by induction on the iteration index $i = 1, 2, \ldots, n$. For clarity, we denote $\mathbb{E}_\bullet(\#\text{ selected in } W'_r)$ as $\mathbb{E}_\bullet(\#)$. For $i = 1$, we have that

$$\begin{aligned}\mathbb{E}_{true} &= \mathbb{E}_{V'[2\ldots n]}\left(\#\middle|V'[1] = true\right) = \mathbb{E}_{X[2\ldots n]}\left(\#\middle|X[1] = true\right),\\ \mathbb{E}_{false} &= \mathbb{E}_{V'[2\ldots n]}\left(\#\middle|V'[1] = false\right) = \mathbb{E}_{X[2\ldots n]}\left(\#\middle|X[1] = false\right).\end{aligned}$$

Given that $\mathbb{E}_{X[1\ldots n]}(\#) = p\mathbb{E}_{X[2\ldots n]}\left(\#\middle|X[1] = true\right) + (1-p)\mathbb{E}_{X[2\ldots n]}\left(\#\middle|X[1] = false\right)$, the claim is true. Now, assuming that the claim is true for iteration $i-1$, we want to prove that $\max\{\mathbb{E}_{true}, \mathbb{E}_{false}\} \geq \mathbb{E}_{X[1\ldots n]}(\#)$ for iteration i, where

$$\begin{aligned}\mathbb{E}_{true} &= \mathbb{E}_{V'[i+1\ldots n]}\left(\#\middle|V'[i] = true\right)\\ \mathbb{E}_{false} &= \mathbb{E}_{V'[i+1\ldots n]}\left(\#\middle|V'[i] = false\right).\end{aligned}$$

By inductive hypothesis we know that

$$\max\{\mathbb{E}_{V'[i\ldots n]}\left(\#\middle|V'[i-1] = true\right), \mathbb{E}_{V'[i\ldots n]}\left(\#\middle|V'[i-1] = false\right)\} \geq \mathbb{E}_{X[1\ldots n]}(\#). \quad (2)$$

Call $\mathbb{E}_{V'[i\ldots n]}(\#)$ the expected number of receivers selected after we fix the value of $V'[i-1]$ in Line 14. That is,

$$\mathbb{E}_{V'[i\ldots n]}(\#) = \max\{\mathbb{E}_{V'[i\ldots n]}\left(\#\middle|V'[i-1] = true\right), \mathbb{E}_{V'[i\ldots n]}\left(\#\middle|V'[i-1] = false\right)\}.$$

Replacing in Equation 2, we have that

$$\mathbb{E}_{V'[i\ldots n]}(\#) \geq \mathbb{E}_{X[1\ldots n]}(\#). \quad (3)$$

We also have that

$$\begin{aligned}\mathbb{E}_{V'[i\ldots n]}(\#) &= p\mathbb{E}_{V'[i+1\ldots n]}\left(\#\middle|V'[i] = true\right) + (1-p)\mathbb{E}_{V'[i+1\ldots n]}\left(\#\middle|V'[i] = false\right)\\ &\leq \max\{\mathbb{E}_{V'[i+1\ldots n]}\left(\#\middle|V'[i] = true\right), \mathbb{E}_{V'[i+1\ldots n]}\left(\#\middle|V'[i] = false\right)\}. \quad (4)\end{aligned}$$

Combining inequalities 4 and 3, the claim follows. ◀