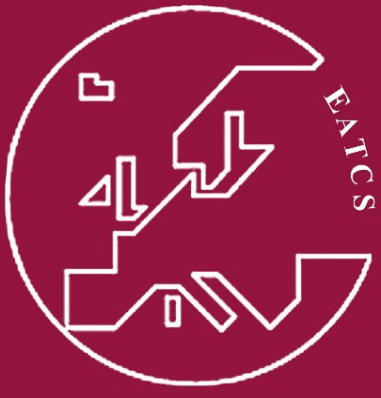
Lars Arge
Michael Hoffmann
Emo Welzl (Eds.)

LNCS 4698

Algorithms – ESA 2007

15th Annual European Symposium
Eilat, Israel, October 2007
Proceedings



Springer

Lecture Notes in Computer Science 4698

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

Editorial Board

Lars Arge Michael Hoffmann Emo Welzl (Eds.)

Algorithms – ESA 2007

15th Annual European Symposium
Eilat, Israel, October 8-10, 2007
Proceedings

Springer

Volume Editors

Lars Arge
University of Aarhus, Department of Computer Science
IT-Parken, Aabogade 34, 8200 Aarhus N, Denmark
E-mail: large@daimi.au.dk

Michael Hoffmann
Emo Welzl
ETH Zurich, Institute for Theoretical Computer Science
Universitätsstr. 6, 8092 Zurich, Switzerland
E-mail: {hoffmann, welzl}@inf.ethz.ch

Library of Congress Control Number: Applied for

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

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

ISSN 0302-9743
ISBN-10 3-540-75519-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-75519-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12170708 06/3180 5 4 3 2 1 0

Preface

This volume contains the 63 contributed papers and abstracts of invited talks presented at the 15th Annual European Symposium on Algorithms (ESA 2007), held in Eilat, Israel during October 8–10, 2007. The three distinguished invited speakers were Pierre Fraigniaud, Christos Papadimitriou and Micha Sharir.

Since 2002, ESA has consisted of two tracks, with Separate Program Committees, which deal with design and mathematical analysis of algorithms, the track, and real-world applications, engineering, and experimental analysis of algorithms, the track. Previous ESAs in the current two-track format were held in Rome, Italy (2002); Budapest, Hungary (2003); Bergen, Norway (2004); Palma de Mallorca, Spain (2005); and Zurich, Switzerland (2006). The proceedings of these symposia were published as Springer's LNCS volumes 2461, 2832, 3221, 3669 and 4168, respectively.

Papers were solicited in all areas of algorithmic research, including Algorithmic Aspects of Networks, Approximation and On-line Algorithms, Computational Biology, Computational Finance and Algorithmic Game Theory, Computational Geometry, Data Structures, Databases and Information Retrieval, External-Memory Algorithms, Streaming Algorithms, Graph and Networks Algorithms, Graph Drawing, Machine Learning, Mobile and Distributed Computing, Pattern Matching and Data Compression, Quantum Computing, Randomized Algorithms, and Algorithm Libraries. The algorithms could be sequential, distributed or parallel. Submissions were especially encouraged in mathematical programming and operations research, including Combinatorial Optimization, Integer Programming, Polyhedral Combinatorics and Network Optimization.

Each extended abstract was submitted to one of the two tracks. The extended abstracts were read by three to four referees each, and evaluated on their quality, originality, and relevance to the symposium. The Program Committees of both tracks met at ETH Zurich during June 1–2, 2007. The Design and Analysis track selected 50 papers out of 165 submissions (one of the accepted papers was a merged version of two submitted papers). The Engineering and Applications track selected 13 out of 44 submissions.

ESA 2007 was sponsored by (the European Association for Theoretical Computer Science). We appreciate the critical financial support of the ALGO 2007 industrial sponsors , , , and . The EATCS sponsorship included an award for the authors of the best paper "Online Primal-Dual Algorithms for Maximizing Ad-Auctions Revenue," by Niv Buchbinder, Kamal Jain, and Joseph (Seffi) Naor, and the best student paper "Order Statistics in the Farey Sequences in Sublinear Time," by Jakub Pawlewicz, as selected by the Program Committees.

July 2007

Lars Arge
Michael Hoffmann
Emo Welzl

Organization

Program Committees

The Program Committees of the two tracks of ESA 2007 consisted of:

Design and Analysis Track

Luca Becchetti	University of Rome "La Sapienza"
Harry Buhrman	CWI and University of Amsterdam
Bruno Codenotti	IIT-CNR Pisa
Gérard P. Cornuéjols	CMU, Pittsburgh and University d'Aix-Marseille
Artur Czumaj	University of Warwick
Leslie Ann Goldberg	University of Liverpool
Edward A. Hirsch	Steklov Inst. of Math., St. Petersburg
Nicole Immorlica	Microsoft Research, Redmond
Satoru Iwata	Kyoto University
Piotr Krysta	University of Liverpool
Zvi Lotker	Ben Gurion University, Beer Sheva
Dániel Marx	Humboldt-University, Berlin
Jiří Matoušek	Charles University, Prague
Seth Pettie	University of Michigan, Ann Arbor
Eric Torng	Michigan State University
Emo Welzl (Chair)	ETH Zurich

Engineering and Applications Track

Lars Arge (Chair)	University of Aarhus
Mark de Berg	TU Eindhoven
Irene Finocchi	University of Rome "La Sapienza"
Mike Goodrich	UC Irvine
Kamesh Munagala	Duke University
Kirk Pruhs	University of Pittsburgh
Rajeev Raman	University of Leicester
Peter Sanders	University of Karlsruhe
Jan Vahrenhold	University of Dortmund
Ke Yi	AT&T Labs Research and HKUST
Christos Zaroliagis	CTI and University of Patras

ESA 2007 was held along with the Fifth Workshop on Approximation and Online Algorithms (WAOA) in the context of the combined conference ALGO 2007. The Organizing Committee of ALGO 2007 consisted of (with special thanks to Nurit Shomrat-Aner):

Yossi Azar	Tel-Aviv University and Microsoft Research
Guy Even	Tel-Aviv University
Amos Fiat (Chair)	Tel-Aviv University
Seffi Naor	Technion and Microsoft Research

Referees

David Abraham
Dimitris Achlioptas
Tatsuya Akutsu
Hesham al-Ammal
Susanne Albers
Noga Alon
Helmut Alt
Ernst Althaus
Christoph Ambühl
Alexandr Andoni
Spyros Angelopoulos
Aaron Archer
Sunil Arya
Nikolaus Augsten
Vincenzo Auletta
Franz Aurenhammer
Giorgio Ausiello
Igor Averbach
Chen Avin
Yossi Azar
Moshe Babaioff
Lars Backstrom
Brenda Baker
Maria-Florina Balcan
Nikhil Bansal
Holger Bast
Michael Bender
Petra Berenbrink
Eric Berberich
Robert Berke
Vincenzo Bonifaci
Allan Borodin
Patrick Briest
Andre Brinkmann
Yves Brise
Gerth Stølting Brodal
Adam Buchsbaum
Andrei Bulatov
Jakub Černý
Ning Chen
Ke Chen
Siu-Wing Cheng
Flavio Chierichetti
Markus Chimani
Marek Chrobak
Andrea Clementi
Amin Coja-Oghlan
Graham Cormode
Jose Correa
Lenore Cowen
Valentino Crespi
Maxime Crochemore
Varsha Dani
Constantinos Daskalakis
Samir Datta
Brian Dean
Xiaotie Deng
Amit Deshpande
Tamal Dey
Martin Dietzfelbinger
Yong Ding
Yefim Dinitz
Benjamin Doerr
Amit Dvir
Khaled Elbassioni
Michael Elkin
Edith Elkind
Leah Epstein
Thomas Erlebach
Tim van Erven
Omid Etesami
Rolf Fagerberg
Martin Farach-Colton
Antonio Faripa
Mohammad Farshi
Márk Félegyházi
Henning Fernau
Jiří Fiala
Matteo Fischetti
Abraham Flaxman
Tamás Fleiner
Lisa Fleischer
Fedor Fomin
Pierre Fraigniaud
Tom Friedetzky
Alan Frieze
Daniele Frigioni
Stefan Funke
Anna Gál
Efstratios Gallopoulos
Julia Gamzova
Sorabh Gandhi
Naveen Garg
Bernd Gärtner
Leszek Gasieniec
Serge Gaspers
Subir Ghosh
Panos Giannopoulos
Fabian Gieseke

Joachim Giesen
Anna Gilbert
Aristides Gionis
Paul Goldberg
Michael Goldwasser
Mordecai Golin
Petr Golovach
Lee-Ad Gottlieb
Vineet Goyal
Fabrizio Grandoni
Tracy Grauman
Martin Grohe
Roberto Grossi
Sudipto Guha
Peter Hachenberger
Mohammad Taghi Hajiaghayi
Dan Halperin
Peter Harremoes
Nick Harvey
Jan van den Heuvel
Michael Hoffmann
Jan Holub
Piotr Indyk
Sandy Irani
Robert Irving
Saitam Issoy
Dmitry Itsykson
Klaus Jansen
Mark Jerrum
Marcin Jurdziński
Crystal Kahn
Lutz Kettner
Philip Klein
Juha Karkkainen
Mark Keil
Steven Kelk
Moshe Lewenstein
David Kempe
Iordanis Kerenidis
Zoltán Király
Itzik Kitroser
Rolf Klein
Adam Klivans
Ton Kloks
Christian Knauer
David B. Knoester
Ina Koch
Arist Kojevnikov
Stavros Kolliopoulos
Roman Kolpakov
Jochen Konemann
Boris Konev
Wouter Koolen-Wijkstra
Guy Kortsarz
Michal Koucký
Darek Kowalski
Daniel Král'
Jan Kratochvíl
Dieter Kratsch
Bhaskar Krishnamachari
Luděk Kučera
Herbert Kuchen
Alexander Kulikov
Abhinav Kumar
Lap Chi Lau
Luigi Laura
Emmanuelle Lebhar
Stefano Leonardi
Nissan Lev-Tov
Meital Levy
Moshe Lewenstein
Ran Libeskind-Hadas
Mathieu Liedloff
Yury Lifshits
Katrina Ligett
Andrzej Lingas
Alex Lopez-Ortiz
Vadim Lozin
Marco Lübbecke
Meena Mahajan
Mohammad Mahdian
Azarakhsh Malekian
Yishay Mansour
Giovanni Manzini
Alberto Marchetti-Spaccamela
Martin Mares
Andrew McGregor
Frank McSherry
Aranyak Mehta
Chad Meiners
Manor Mendel
Julian Mestre
Adam Meyerson
Dezső Miklós
István Miklós
Kevin Milans
Vahab Mirrokni
Cristopher Moore
Dmitriy Morozov
Marcin Mucha
Boris Naujoks
Gonzalo Navarro
Hamid Nazerzadeh
Rolf Niedermeier
Sergey Nikolenko
Evdokia Nikolova
Yoshio Okamoto
Martin Pál
Panagiota Panagopoulou
Alessandro Panconesi
Mihai Patrascu
Boaz Patt-Shamir
Wolfgang Paul
David Pearce
Mark Pedigo
Marco Pellegrini
Sriram Pemmaraju
Paolo Penna
Giuseppe Persiano
Attila Pethő
Andrea Pietracaprina
Sylvain Pion
Greg Plaxton
Laura Pozzi
Grigorios Prasinos
Yuri Pritykin
Kirk Pruhs
Harald Räcke
Luis Rademacher
Tomasz Radzik
Mathieu Raffinot
Daniel Raible
Dror Rawitz

Andreas Razen
Igor Razgon
Liam Roditty
Amir Ronen
Adi Rosen
Gianluca Rossi
Tim Roughgarden
Frank Ruskey
Daniel Russel
Wojciech Rytter
Mohammad Reza Salavatipour
Piotr Sankowski
Paolo Santi
Srinivasa Rao Satti
Saket Saurabh
Rahul Savani
Joe Sawada
Nitin Saxena
Gabriel Scalosub
Thomas Schank
Dominik Scheder
Stefan Schirra
Alex Scott
Michael Segal
Danny Segev
Hadas Shachnai
Moni Shahar
Asaf Shapira
Igor Shparlinski
Arseny Shur
Anastasios Sidiropoulos
Alain Sigayret
Johannes Singler
Amitabh Sinha
Carsten Sinz
Rene Sitters
Michiel Smid
Shakhar Smorodinsky
Jack Snoeyink
Christian Sohler
Alexander Souza
Robert Spalek
Frits Spieksma
Aravind Srinivasan
Matthias Stallmann
Rob van Stee
Daniel Štefankovič
Bernhard von Stengel
Miloš Stojaković
Martin Strauss
Dirk Sudholt
Marek Sulovský
Ozgur Sumer
Xiaoming Sun
Ravi Sundaram
Mukund Sundararajan
Eric Tannier
Sachio Teramoto
Thorsten Theobald
Shripad Thite
Mikkel Thorup
Alexander Tiskin
Isaac To
Ben Toner
Mark R. Tuttle
Ryuhei Uehara
Falk Unger
Takeaki Uno
Kasturi Varadarajan
Sergei Vassilvitskii
Santosh Vempela
Rossano Venturini
Florian Verhein
Adrian Vetta
Stéphane Vialette
Berthold Vöcking
Jan Vondrák
Tjark Vredeveld
Uli Wagner
Xin Wang
Bo Wang
Ryan Williams
Paul Wollan
Nicola Wolpert
Prudence Wong
David Wood
Ke Yi
Raphael Yuster
Mohammed Zaki
Hairong Zhao
Florian Zickfeld
Michele Zito
Afra Zomorodian
Philipp Zumstein

Table of Contents

Invited Lectures

Contributed Papers: Design and Analysis Track

Contributed Papers: Engineering and Applications Track

Nash Equilibria: Where We Stand

Christos H. Papadimitriou*

Computer Science Division, UC Berkeley

Abstract. In the Fall of 2005 it was shown that finding an ϵ-approximate mixed Nash equilibrium in a normal-form game, even with two players, is PPAD-complete for small enough (additive) ϵ — and hence, presumably, an intractable problem. This solved a long-standing open problem in Algorithmic Game Theory, but created many open questions. For example, it is known that inverse polynomial ϵ is enough to make the problem intractable, while, for two player games, relatively simple polynomial algorithms are known to achieve ϵ near $\frac{1}{3}$; bridging this gap is an important open problem.

When the number of strategies per player is small, a different set of algorithmic techniques comes into play; it had been known, for example, that *symmetric* games of this sort can be solved in polynomial time, via a reduction to the existential theory of the reals. In on-going joint work with Costis Daskalakis we have shown that a simple exhaustive approach works in a broader, and more useful in practice, class of games known as *anonymous* games, in which the payoff of each player and strategy is a symmetric function of the strategies chosen by the other players; that is, a player's utility depends on *how many* other players have chosen each of the strategies, and not on *precisely which players* have. In fact, a variant of the same algorithmic technique gives a pseudopolynomial-time approximation scheme for general n-player games, as long as the number of strategies is kept a constant. Improving this to polynomial seems a challenging problem.

A third important front in this research project is exploring equilibrium concepts that are more attractive computationally than the mixed Nash equilibrium, and possibly more natural, yet no less universal (guaranteed to exist under quite general assumptions). A number of such alternatives have been explored recently, some of them in joint work with Alex Fabrikant. For example, we show that two-player games with random entries of the utility matrices are likely to have a natural generalization of a pure Nash equilibrium called *unit recall equilibrium.*

Finally, it had long been believed that Nash equilibria of *repeated games* are much easier to find, due to a cluster of results known in Game Theory as *the Folk Theorem.* We shall discuss how recent algorithmic insights cast doubt even to this reassuring fact.

* Research partially supported by NSF grant CCF-0635319, a MICRO grant, and a research gift from Yahoo! Research.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, p. 1, 2007.

Small Worlds as Navigable Augmented Networks: Model, Analysis, and Validation

Pierre Fraigniaud*

CNRS and University of Paris 7

Abstract. The small world phenomenon, a.k.a. the six degree of separation between individuals, was identified by Stanley Milgram at the end of the 60s. Milgram experiment demonstrated that letters from arbitrary sources and bound to an arbitrary target can be transmitted along short chains of closely related individuals, based solely on some characteristics of the target (professional occupation, state of leaving, etc.). In his paper on small world navigability, Jon Kleinberg modeled this phenomenon in the framework of augmented networks, and analyzed the performances of greedy routing in augmented multi-dimensional meshes. This paper objective is to survey the results that followed up Kleinberg seminal work, including results about:

- extensions of the augmented network model, and variants of greedy routing,
- designs of polylog-navigable graph classes,
- the quest for universal augmentation schemes, and
- discussions on the validation of the model in the framework of doubling metrics.

1 Greedy Routing in Augmented Meshes

1.1 Modeling Milgram Experiment

Augmented graphs were introduced in [25,26] for the purpose of understanding the small world phenomenon. Precisely, augmented graphs give a framework for modeling and analyzing the "six degrees of separation" between individuals observed from Milgram experiment [37], stating that short chains of acquaintances between any pair of individuals can be discovered in a distributed manner. Note that the augmented graph model addresses [illegible] in small worlds, and therefore it goes further than just structural considerations (e.g., scale free properties, clustering properties, etc.).

We define an [illegible] [15] as a pair (G, φ) where G is a graph, called [illegible], and φ is a probability distribution, referred to as an [illegible] for G. This augmenting distribution is defined as a collection of probability distributions $\{\varphi_u, u \in V(G)\}$. Every node $u \in V(G)$ is given few (typically one) extra links, called [illegible], pointing to some nodes, called the [illegible] of u. Any long range contact v is chosen at random

* Additional supports from COST Action 295 "DYNAMO".

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 2–11, 2007.

according to $\Pr\{u \to v\} = \varphi_u(v)$. (If $v = u$ or v is a neighbor of u in G, then no link is added). In this paper, a graph in (G, φ) will sometime be denoted by $G + L$ where G is the base graph, and L is the set of long rang links resulting from the trial of φ yielding that set of links added to G.

A important feature of this model is that it enables to define simple but efficient decentralized routing protocols modeling the search procedure applied by social entities in Milgram [37] and Dodd et al [9] experiments. In particular, [illegible] in a graph in (G, φ) is the oblivious routing process in which every intermediate node along a route from a source $s \in V(G)$ to a target $t \in V(G)$ chooses among all its neighbors (including its long range contacts) the one that is the closest to t [illegible] G, and forwards to it. For this process to apply, the only "knowledge" that is supposed to be available at every node is its distances to the other nodes in the base graph G. This assumption is motivated by the fact that, if the base graph offers some nice properties (e.g., embeddable in a low dimensional metric with small distorsion) then the distance function $dist$ in G is expected to be easy to compute, or at least to approximate, locally. In Milgram experiment, one may think about G as representing the geography of the earth.

The [illegible] of (G, φ) is defined as

$$\mathrm{diam}(G, \varphi) = \max_{s,t \in V(G)} \mathbb{E}(\varphi, s, t)$$

where $\mathbb{E}(\varphi, s, t)$ is the expected number of steps for traveling from s to t using greedy routing in (G, φ). For each pair s, t, the expectation is computed over all trials of φ, i.e., over all graphs in (G, φ).

In [25], Kleinberg considered the n-node d-dimensional square meshes with wraparound links, that we denote by $\mathcal{M}_n^{(d)}$, augmented using the d-harmonic distribution $h^{(d)}$ defined as follows:

$$h_u^{(d)}(v) = \frac{1}{Z_u} \frac{1}{dist(u,v)^d} \text{ where } Z_u = \sum_{v \neq u} \frac{1}{dist(u,v)^d}.$$

For $d = 2$ this setting expresses the fact that two individuals are more likely to be connected if they leave nearby, and that their probability of being connected decreases inversely proportional to their geographical distance square. For $d > 2$, the extra dimensions could be interpreted as as many criteria (professional occupation, hobbies, etc.). [25] proved that the greedy diameter of $(\mathcal{M}_n^{(d)}, h^{(d)})$ is $O(\log^2 n)$. Hence, greedy routing performs in a polylogarithmic expected number of steps by just adding one extra link per node. [25] also established an interesting threshold phenomenon: if $d' \neq d$, then the greedy diameter of $(\mathcal{M}_n^{(d)}, h^{(d')})$ is $\Omega(n^\epsilon)$ for some $\epsilon > 0$ depending on d/d'. Hence for $d' = d$ the greedy diameter is polylogarithmic whereas it is polynomial for $d \neq d'$. The model is therefore very sensitive to the distance scaling for the choices of the long range contacts.

It is worth mentioning that the augmented mesh model was used in [32] for the design of an overlay network supporting the DHT facilities for P2P systems. See also [36] for an application to network design. More generally, we refer to [28]

for a survey of Jon Kleinberg contributions to the analysis of complex networks, including the contribution [27] describing a hierarchical model for the small world phenomenon.

1.2 Extension of the Model

Numerous contributions immediately followed up Kleinberg seminal work, all aiming at generalizing the routing strategy in augmented multidimensional meshes. In particular, [7,33] proved that if every node knows not only its long range contacts but also the long range contacts of its neighbors, then greedy routing in $(\mathcal{M}_n^{(d)}, h^{(d)})$ with $\log n$ long range links per node performs in $O(\log n / \log\log n)$ expected number of steps, i.e., the greedy diameter becomes as small as the expected diameter of the augmented mesh with $\log n$ long range links per node. See [35] for more information on the (standard) diameter of $(\mathcal{M}_n^{(d)}, h^{(d)})$.

[34] somehow generalized this approach by assuming that every node knows the long range links of its $\log n$ closest neighbors in the mesh, and proposed a non oblivious routing protocol performing in $O(\log^{1+1/d} n)$ expected number of steps in this context. The same model was simultaneously proposed in [16,17] where an oblivious greedy routing protocol is proposed, performing in $O(\frac{1}{k}\log^{1+1/d} n)$ expected number of steps with k long range contacts per node. [16] summarized their result as "eclecticism shrinks even small worlds" because their result can be interpreted as demonstrating the impact of the dimension d of the world on the efficiency of routing. If this dimension merely reflects the number of criteria used for routing, then it is better having few acquaintances (i.e., few long range contacts) of many different nature (i.e., routing in a high dimensional world) than having many acquaintances (i.e., many long range contacts) of similar profile (i.e., routing in a low dimensional world). See [24] for experimental results about how individuals are impacted by the number of criteria for the search.

[31] investigated a sophisticated decentralized algorithm that visits $O(\log^2 n)$ nodes, and distributively discovers routes of expected length $O(\log n\ (\log\log n)^2)$ links using headers of size $O(\log^2 n)$ bits. This algorithm could be used in, e.g., a P2P system in which resources would be sent along the reverse path used by the request to reach its target: reaching the target requires $O(\log^2 n)$ expected number of steps but the resource would follow a much shorter route, of length $O(\log n\ (\log\log n)^2)$ links.

1.3 Lower Bounds

[5] proved that the analysis in [25] is tight in the sense that greedy routing in directed cycles augmented with the 1-harmonic distribution performs in $\Omega(\log^2 n)$ expected number of steps. This result is generalized in [16] for any $d \geq 1$. There are augmenting distributions $\varphi^{(d)}$ that do better than the d-harmonic distribution $h^{(d)}$ in d-dimensional meshes. For instance, [13] designed augmenting distributions $\varphi^{(d)}$ for $\mathcal{M}_n^{(d)}$ for any $d \geq 1$, such that the greedy diameter of $(\mathcal{M}_n^{(d)}, \varphi^{(d)})$ is $O(\log n)$. (See also [2] for the case of the ring). However, if the

probability distribution is bounded to decrease with the distance (a natural assumption as far as social networks are concerned), then it is still open whether one can do better than the harmonic distribution. Several papers [4,13,21] actually tackled the problem of proving that, for any non-increasing distribution $\varphi^{(d)}$, greedy routing in $(\mathcal{M}_n^{(d)}, \varphi^{(d)})$ performs in $\Omega(\log^2 n)$ expected number of steps. Essentially, they proved that $\Omega(\log^2 n)$ is indeed a lower bound for greedy routing in rings, but only up to $\log\log n$ factors. With $\log n$ long range links per node, [33] proved a lower bound $\Omega(\log n)$ for any 1-local algorithm, in contrast to the 2-local Neighbor-of-Neighbor algorithm [7,33] that performs in $O(\log n/\log\log n)$ expected number of steps.

2 Graphs with Polylogarithmic Greedy Diameter

Another trend of researches aimed at considering the augmented graph model for base graphs different from meshes. Given a graph class $\mathcal{G}$, the objective is to design augmenting distributions satisfying that, for any $G \in \mathcal{G}$, there exists φ such that (G, φ) has small greedy diameter, typically polylog(n). The number of long range contacts per node is preferably 1, but any polylog(n) number is fine. Actually, the ability to add k links instead of 1 generally leads to a speed up of $\Theta(k)$ in greedy routing, and thus does not modify the essence of the problem whenever $k \leq$ polylog(n).

In $\mathcal{M}_n^{(d)}$, the d-harmonic distribution satisfies that for any two nodes u and v at distance r, the probability that u has v as long rang contact is proportional to $1/r^d$, that is inversely proportional to the size of the ball centered at u and of radius r. This leads several groups to investigate graph classes for which the augmenting distribution satisfies $\varphi_u(v) \propto 1/|B_u(r)|$, where $r = dist(u, v)$ and $B_u(r)$ is the ball centered at u of radius r. A graph G has ball growth α if, for any $r > 0$, the size of any ball of radius r in G is at most α times the size of the ball of radius $r/2$ centered at the same node. [10] proved that graphs of bounded ball growth can be augmented such that their greedy diameter becomes polylogarithmic. ([11] even proves that the augmenting process can be performed efficiently in a distributed manner). [41] extended this result to graphs of bounded doubling dimension. In fact, this latter paper even considered general metrics, not only graph metrics. A metric space is of doubling dimension α if any ball of radius r in S can be covered by 2^α balls of radius $r/2$. [41] proved that metrics of bounded doubling dimension can be augmented such that their greedy diameter becomes polylogarithmic (see also [22]). Actually, the result holds even for metrics with doubling dimension $\Theta(\log\log n)$.

In a somewhat orthogonal direction, [13] considered trees, and proved that any tree can be augmented so that the resulting greedy diameter becomes $O(\log n)$. This result was extended to graphs of bounded treewidth. A tree-decomposition of a graph G is a pair $(T, \mathcal{X})$ where T is a tree of vertex set I, and $\mathcal{X} = \{X_i, i \in I\}$ is a collection of "bags" $X_i \subseteq V(G)$. This pair must satisfy the following:

- $\cup_{i \in I} X_i = V(G)$;
- for any edge $\{u, v\} \in E(G)$ there exists $i \in I$ such that $\{u, v\} \subseteq X_i$; and
- for any node $u \in V(G)$, the set $\{i \in I, u \in X_i\}$ is a subtree of T.

The width of $(T, \mathcal{X})$ is $\max_i |X_i| - 1$, and the treewidth of G is the minimum width of any tree-decomposition of G. Treewidth is an important concept in the core of the Graph Minor theory [40], and with many applications to the design of fixed parameter algorithms. On graphs of treewidth at most k, where k is fixed, every decision or optimization problem expressible in monadic second-order logic has a linear algorithm [8]. As far as navigability in small worlds is concerned, this notion is also appealing since a tree-decomposition of the base graph G representing the predictable acquaintances between the individuals determines a hierarchy between these individuals that is inherited from these acquaintances, and not specified a priori as in [27]. [14] proved that graphs of bounded treewidth can be augmented such that their greedy diameter becomes polylogarithmic. This result was in turn generalized to graphs excluding a fixed minor. A graph H is a minor of a graph G if H can be obtained from G by a sequence of operations edge contraction, edge deletion, or node deletion. [1] proved that for any fixed H, all H-minor free graphs can be augmented such that their greedy diameter becomes polylogarithmic. This result is actually one among the many consequences of a more general result on path separability by [1], using the characterization of H-minor free graphs by Robertson and Seymour.

Polylog-navigability also holds for many graph classes that are not included in the aforementioned classes. This is the case of interval graphs and AT-free graphs [15].

Finally, it is worth mentioning [29] which developed a variant of the model with population density varying across the nodes of a network G with bounded doubling dimension (every node contains a certain number of people, like a village, a town, or a city). It proposed a rank-based augmenting distribution φ where $\varphi_u(v)$ is inversely proportional to the number of people who are closer to u than v is. ([30] indeed showed that in some social networks, two-third of the friendships are actually geographically distributed this way: the probability of befriending a particular person is inversely proportional to the number of closer people). [29] proved that the greedy diameter of (G, φ) is polylogarithmic. Note however that routing must not reach a target, but simply the population cluster in which the target is located.

3 Universal Augmentation Schemes

[13] proved that deciding whether, given a base graph G as input, there exists an augmenting distribution φ such that the greedy diameter of (G, φ) is at most 2 is NP-complete. Still, the previous section surveyed a large number of graph classes, of different nature, that can be all augmented so that their greedy diameter becomes polylogarithmic. It is however not true that all graphs can be augmented to get polylogarithmic greedy diameter.

Let $f : \mathbb{N} \to \mathbb{R}$ be a function. An n-node graph G is f [illegible] if there exists a collection of probability distributions φ such that $\mathrm{diam}(G, \varphi) \leq f(n)$. [18] proved that a function f such that every n-node graph is f-navigable satisfies $f(n) = \Omega(n^{1/\sqrt{\log n}})$. Surprisingly, the graphs G_n used to prove this result are very regular. G_n is the graph of n nodes consisting of p^k nodes labeled $(x_1, \ldots, x_k)$, $x_i \in \mathbf{Z}_p$. Node $(x_1, \ldots, x_k)$ is connected to all nodes $(x_1 + a_1, \ldots, x_k + a_k)$ where $a_i \in \{-1, 0, 1\}$, $i = 1, \ldots, k$, and all operations are taken modulo p. By construction of G_n, the distance between two nodes $y = (y_1, \ldots, y_k)$ and $z = (z_1, \ldots, z_k)$ is $\max_{1 \leq i \leq k} \min(|y_i - z_i|, p - |y_i - z_i|)$. Hence, the diameter of G_n is $\lfloor p/2 \rfloor$. Its doubling dimension is k. Thus, intuitively, a graph G_n has too many "directions", precisely 2^k directions, and it takes at least 2^k expected number of steps to go in the right direction. Therefore, if $k \gg \log \log n$, then the expected number of steps of greedy routing is not polylogarithmic. The result in [41] is therefore essentially the best that can be achieved by considering only the doubling dimension of graphs. Solving $2^k = p = n^{1/k}$ yields $k = \sqrt{\log n}$, and the expected diameter of (G_n, φ) is $\Omega(n^{1/\sqrt{\log n}})$ for this setting of k, for any φ.

On the other hand, Peleg [38] noticed that any n-node graph is $O(\sqrt{n})$-navigable by adding long-range links whose extremities are chosen uniformly at random among all the nodes in the graph. To see why, consider the ball B of radius $\sqrt{n}$ centered at the target. The expected number of nodes visited until the long range contact of the current node belongs to B is $n/|B|$, and thus at most $\sqrt{n}$. Once in B, the distance to the target is at most $\sqrt{n}$. Hence the $O(\sqrt{n})$-navigability of the graph. The best upper bound known so far is actually $n^{1/3} \cdot \mathrm{polylog}(n)$, due to [15]. The augmentation consists for every node in choosing a "level" $i \in \{1, \ldots, \log n\}$, and then selecting the long range contact uniformly at random in the ball centered at the node and of radius 2^i.

4 Validating the Augmented Graph Model

The augmented graph framework is now well understood, and it is probably time to consider the validation of the model, for instance by performing experiments on social networks. This however requires the design of algorithms that, given a graph $G = H + L$ as input, would be able to separate the base graph H from the long range links L. An attempt in this direction has been proposed in [3,6]. In these two papers the authors introduce an hybrid model that resembles the augmented graph model, defined by a base graph H and a global graph L, over the same set of vertices. In both papers, L is a random power law graph. In [6], the base graph H has a local connectivity characterized by a certain number of edge-disjoint paths of bounded length connecting the two extremities of any edge. In [3], the local connectivity is characterized by an amount of flow that can be pushed from one extremity of an edge to the other extremity, along routes of bounded length. In both cases, in addition to presenting a set of informative results about their models, the authors give an algorithm that can recover the base graph almost perfectly. Nevertheless, navigability is not considered in [3,6].

In [19], it is proved that if G has a clustering coefficient $\Omega(\log n/\log\log n)$, and bounded doubling dimension, then a simple algorithm enables to partition the edges of $G = H + L$ into two sets H' and L' if L is obtained by the augmenting distributions in [10] or [41]. The set H' satisfies $E(H) \subseteq H'$ and the edges in $H' \setminus E(H)$ are of small stretch, i.e., the map H is not perturbed too greatly by undetected long range links remaining in H'. The perturbation is actually so small that it is proved that the expected performances of greedy routing in G using the distances in H' are close to the expected performances of greedy routing in $H + L$. Although this latter result may appear intuitively straightforward, since $H' \supseteq E(H)$, it is not, as it is also shown that routing with a map more precise than H may actually damage greedy routing significantly. It is also shown that in absence of a hypothesis regarding the clustering coefficient, any structural attempt to extract the long range links will miss the detection of at least $\Omega(n^{5\varepsilon}/\log n)$ long range links of stretch at least $\Omega(n^{1/5-\varepsilon})$ for any $0 < \varepsilon < 1/5$, and thus the base graph H cannot be recovered with good accuracy.

The extraction algorithm in [19] applies to graphs with bounding ball growth, or with even just bounded doubling dimension. The bounded ball growth assumption can be well motivated intuitively [39] and is consistent with the transit-stub model [42]. The formal verification of the bounded ball growth assumption has been statistically established by [12,43], but only in average. Recently, [20], analyzes the distance defined in the Internet by the round-trip delay (RTT). Based on skitter data collected by CAIDA, it is noted that the ball growth of the Internet, as well as its doubling dimension, can actually be quite large. Nevertheless, it is observed that the doubling dimension is much smaller when restricting the measures to balls of large enough radius (like in, e.g., [23]). Also, by computing the number of balls of radius r required to cover balls of radius $R > r$, it is observed that this number grows with R much slower than what is predicted by a large doubling dimension. However, based on data collected on the PlanetLab platform, it is confirmed that the triangle inequality does not hold for a significant fraction of the nodes. Nevertheless, it is demonstrated that RTT measures satisfy a weak version of the triangle inequality: there exists a small constant ρ such that for any triple u, v, w, we have $RTT(u, v) \leq \rho \cdot \max\{RTT(u, w), RTT(w, v)\}$. (Smaller bounds on ρ can even be obtained when the triple u, v, w is skewed). Based on these observations, [20] proposes an analytical model for Internet latencies. This model is tuned by a small set of parameters concerning the violation of the triangle inequality and the geometrical dimension of the network, and is proved tractable by designing a simple and efficient compact routing scheme with low stretch. The same techniques as the one used for designing this scheme could be used for graph augmentation.

5 Further Works

We identify three directions for further research in the field of small world navigability (see [28] for a more general vision of the problem). One crucial problem

is to close the gap between the upper bound $n^{1/3} \cdot \text{polylog}(n)$ and the lower bound $\Omega(n^{1/\sqrt{\log n}})$ for navigability in arbitrary graphs. Reaching the lower bound seems to be challenging if not relaxing the routing strategy. Greedy routing is indeed actually a very strong constraint. In particular, it does not permit any detour via hubs that could enable discovering shorter routes.

Hierarchical models such as the ones in [27] and [14] suggest that it may be possible to extend the theory of navigability to matroids. In the matroid context, many tools were recently developed, including branch-decomposition, that could enable generalizing the theory for graphs to matroids. For instance, the potential function guiding greedy routing could be defined as the height of the lowest common ancestor in an optimal branch decomposition.

Finally, one is still very far from a complete validation of the augmented graph model for social networks. The results in [3,6,19] tell us that the design of algorithms extracting the long-range links is doable, at least for some large classes of augmented graphs. However, the design of a practical algorithm that could be run on real samples of social networks remains to be done. This is a challenging and rewarding task.

The author wants to express his gratefulness to his colleagues and friends Cyril Gavoille, Emmanuelle Lebhar, and Zvi Lotker for all the fun and great time he has while working with them, especially on the topic of this paper.

References

1. Abraham, I., Gavoille, C.: Object Location Using Path Separators. In: 25th ACM Symp. on Principles of Distributed Computing (PODC), pp. 188–197. ACM Press, New York (2006)
2. Abraham, I., Malkhi, D., Manku, G.: Brief Announcement: Papillon: Greedy Routing in Rings. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 514–515. Springer, Heidelberg (2005)
3. Andersen, R., Chung, F., Lu, L.: Modeling the small-world phenomenon with local network flow. Internet Mathematics 2(3), 359–385 (2006)
4. Aspnes, J., Diamadi, Z., Shah, G.: Fault-tolerant routing in peer-to-peer systems. In: 21st ACM Symp. on Principles of Distributed Computing (PODC), pp. 223–232. ACM Press, New York (2002)
5. Barrière, L., Fraigniaud, P., Kranakis, E., Krizanc, D.: Efficient Routing in Networks with Long Range Contacts. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 270–284. Springer, Heidelberg (2001)
6. Chung, F., Lu, L.: The small world phenomenon in hybrid power law graphs. Lect. Notes Phys. 650, 89–104 (2004)
7. Coppersmith, D., Gamarnik, D., Sviridenko, M.: The Diameter of a Long-Range Percolation Graph. Random Structures and Algorithms 21(1), 1–13 (2002)
8. Courcelle, B., Makowsky, J., Rotics, U.: Linear-time solvable optimization problems on graphs of bounded cliquewidth. In: Hromkovič, J., Sýkora, O. (eds.) Graph-Theoretic Concepts in Computer Science. LNCS, vol. 1517, pp. 1–16. Springer, Heidelberg (1998)

9. Dodds, P., Muhamad, R., Watts, D.: An experimental study of search in global social networks. Science 301(5634), 827–829 (2003)
10. Duchon, P., Hanusse, N., Lebhar, E., Schabanel, N.: Could any graph be turned into a small-world? Theoretical Computer Science 355(1), 96–103 (2006)
11. Duchon, P., Hanusse, N., Lebhar, E., Schabanel, N.: Towards small world emergence. In: 18th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 225–232. ACM Press, New York (2006)
12. Faloutsos, M., Faloutsos, P., Faloutsos, C.: On Power-law Relationships of the Internet Topology. In: ACM SIGCOMM Conference, pp. 251–262. ACM Press, New York (1999)
13. Flammini, M., Moscardelli, L., Navarra, A., Perennes, S.: Asymptotically optimal solutions for small world graphs. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 414–428. Springer, Heidelberg (2005)
14. Fraigniaud, P.: Greedy routing in tree-decomposed graphs: a new perspective on the small-world phenomenon. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 791–802. Springer, Heidelberg (2005)
15. Fraigniaud, P., Gavoille, C., Kosowski, A., Lebhar, E., Lotker, Z.: Universal Augmentation Schemes for Network Navigability: Overcoming the $\sqrt{n}$-Barrier. In: 19th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA), ACM Press, New York (2007)
16. Fraigniaud, P., Gavoille, C., Paul, C.: Eclecticism shrinks even small worlds. In: 23rd ACM Symposium on Principles of Distributed Computing (PODC), pp. 169–178. ACM Press, New York (2004)
17. Fraigniaud, P., Gavoille, C., Paul, C.: Eclecticism shrinks even small worlds. Distributed Computing 18(4) (2006)
18. Fraigniaud, P., Lebhar, E., Lotker, Z.: A doubling dimension threshold $\Theta(\log \log n)$ for augmented graph navigability. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 376–386. Springer, Heidelberg (2006)
19. Fraigniaud, P., Lebhar, E., Lotker, Z.: Recovering the Long Range Links in Augmented Graphs. Technical Report 6197, INRIA Paris-Rocquencourt (May 2007)
20. Fraigniaud, P., Lebhar, E., Viennot, L.: The Inframetric Model for the Internet. Technical Report, INRIA Paris-Rocquencourt (July 2007)
21. Giakkoupis, G., Hadzilacos, V.: On the complexity of greedy routing in ring-based peer-to-peer networks. In: 26th ACM Symposium on Principles of Distributed Computing (PODC), ACM Press, New York (2007)
22. Har-Peled, S., Mendel, M.: Fast Construction of Nets in Low Dimensional Metrics, and Their Applications. SIAM J. on Computing 35(5), 1148–1184 (2006)
23. Karger, D., Ruhl, M.: Finding nearest neighbors in growth-restricted metrics. In: 34th ACM Symp. on the Theory of Computing (STOC), pp. 63–66. ACM Press, New York (2002)
24. Killworth, P., Bernard, H.: Reverse Small-World Experiment. Social Networks 1(2), 159–192 (1978)
25. Kleinberg, J.: The Small-World Phenomenon: An Algorithmic Perspective. In: 32nd ACM Symp. on Theory of Computing (STOC), pp. 163–170. ACM Press, New York (2000)
26. Kleinberg, J.: Navigation in a Small-World. Nature 406, 845 (2000)
27. Kleinberg, J.: Small-World Phenomena and the Dynamics of Information. In: 15th Neural Information Processing Systems (NIPS) (2001)
28. Kleinberg, J.: Complex networks and decentralized search algorithm. Nevanlinna prize presentation at the International Congress of Mathematicians (ICM), Madrid (2006)

29. Kumar, R., Liben-Nowell, D., Tomkins, A.: Navigating Low-Dimensional and Hierarchical Population Networks. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, Springer, Heidelberg (2006)
30. Liben-Nowell, D., Novak, J., Kumar, R., Raghavan, P., Tomkins, A.: Geographic routing in social networks. In: Proc. of the Natl. Academy of Sciences of the USA, vol. 102/3, pp. 11623–11628 (2005)
31. Lebhar, E., Schabanel, N.: Searching for Optimal paths in long-range contact networks. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 894–905. Springer, Heidelberg (2004)
32. Manku, G., Bawa, M., Raghavan, P.: Symphony: distributed hashing in a small world. In: 4th USENIX Symp. on Internet Technologies and Systems, pp. 127–140 (2003)
33. Manku, G., Naor, M., Wieder, U.: Know Thy Neighbor's Neighbor: The Power of Lookahead in Randomized P2P Networks. In: 36th ACM Symp. on Theory of Computing (STOC), pp. 54–63. ACM Press, New York (2004)
34. Martel, C., Nguyen, V.: Analyzing Kleinberg's (and other) Small-world Models. In: 23rd ACM Symp. on Principles of Distributed Computing (PODC), pp. 179–188. ACM Press, New York (2004)
35. Martel, C., Nguyen, V.: Analyzing and characterizing small-world graphs. In: 16th ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 311–320. ACM Press, New York (2005)
36. Martel, C., Nguyen, V.: Designing networks for low weight, small routing diameter and low congestion. In: 25th Conference of the IEEE Communications Society (INFOCOM), IEEE Computer Society Press, Los Alamitos (2006)
37. Milgram, S.: The Small-World Problem. Psychology Today, 60–67 (1967)
38. Peleg, D.: Private communication. Workshop of COST Action 295 "DYNAMO", Les Ménuires (January 2006)
39. Plaxton, G., Rajaraman, R., Richa, A.: Accessing Nearby Copies of Replicated Objects in a Distributed Environment. Theory of Computing Systems 32(3), 241–280 (1999)
40. Robertson, N., Seymour, P.D.: Graph minors II, Algorithmic Aspects of Tree-Width. Journal of Algorithms 7, 309–322 (1986)
41. Slivkins, A.: Distance estimation and object location via rings of neighbors. In: 24th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 41–50. ACM Press, New York (2005)
42. Zegura, E., Calvert, K., Bhattacharjee, S.: How to Model an Internetwork. In: 14th Conference of the IEEE Computer and Communications Societies (INFOCOM), pp. 594–602. IEEE Computer Society Press, Los Alamitos (1996)
43. Zhang, B., Ng, T., Nandi, A., Riedi, R., Druschel, P., Wang, G.: Measurement based analysis, modeling, and synthesis of the internet delay space. In: 6th Internet Measurement Conference (IMC), pp. 85–98 (2006)

Arrangements in Geometry: Recent Advances and Challenges*

Micha Sharir

School of Computer Science, Tel Aviv University, Tel Aviv, Israel
michas@post.tau.ac.il

Abstract. We review recent progress in the study of arrangements in computational and combinatorial geometry, and discuss several open problems and areas for further research.

In this talk I will survey several recent advances in the study of arrangements of curves and surfaces in the plane and in higher dimensions. This is one of the most basic structures in computational and combinatorial geometry. Arrangements appear in a variety of application areas, such as geometric optimization, robotics, graphics and modelling, and molecular biology, just to name a few. Arrangements also possess their own rich structure, which has fueled extensive research for the past 25 years (although, if one wishes, one can find the first trace of them in a study by Steiner in 1826 [37]). While considerable progress has been made, it has left many "hard nuts" that still defy a solution. The aim of this talk is to present these difficult problems, describe what has been done, and what are the future challenges.

An arrangement of a collection S of n surfaces in $\mathbb{R}^d$ is simply the decomposition of d-space obtained by "drawing" the surfaces. More formally, it is the decomposition of d-space into maximal relatively open connected sets, of dimension $0, 1, \ldots, d$, where each set ("face") is contained in the intersection of a fixed subset of the surfaces, and avoids all other surfaces. In many applications, one is interested only in certain substructure of the arrangement, such as lower envelopes, single cells, union of regions, levels, and so on. Other applications study certain constructs related to arrangements, such as incidences between points and curves or surfaces, or cuttings and decompositions of arrangements.

The topics that the talk will aim to address (and, most likely, only partially succeed) include:

(a) Union of geometric objects: In general, the maximum combinatorial complexity of the union of n simply shaped objects in $\mathbb{R}^d$ is $\Theta(n^d)$. However, in many favorable instances better bounds can be established, include unions of fat objects and unions of Minkowski sums of certain kinds; in most cases,

* Work on this paper was partially supported by NSF Grant CCF-05-14079, by a grant from the U.S.-Israeli Binational Science Foundation, by grant 155/05 from the Israel Science Fund, Israeli Academy of Sciences, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 12–16, 2007.

these bounds are close to $O(n^{d-1})$, which is asymptotically tight. I will briefly review the significant recent progress made on these problems, and list the main challenges that still lie ahead. The main open problems involve unions in three and higher dimensions. For more details, see a recent survey by Agarwal et al. [3].

(b) Decomposition of arrangements: In many algorithmic and combinatorial applications of arrangements, one uses divide-and-conquer techniques, in which the space is decomposed into a small number of regions, each of which is crossed by only a small number of the n given curves or surfaces. Ideally, for a specified parameter r, one seeks a decomposition (also known as a $(1/r)$-*cutting*) into $O(r^d)$ regions, each crossed by at most n/r of the curves or surfaces. This goal has been achieved for planar arrangements, and for arrangements of hyperplanes in any dimesnion. For general simply-shaped surfaces in dimensions three and four, there exist $(1/r)$-cuttings of size close to $O(r^d)$. The problem is wide open in five and higher dimensions. Several (hard) related problems, such as complexity of the overlay of minimization diagrams, or of the sandwich region between two envelopes, will also be mentioned. There is in fact only one method for decomposing arrangements of semi-algebraic surfaces, which is the *vertical decomposition* (see [13] and the many references given below), and the challenge is to understand its maximum combinatorial complexity. For more details, see the book [35], and several surveys on arrangements [5, 6, 34].

(c) Incidences between points and curves and related problems: Bounding the number of incidences between m distinct points and n distinct curves or surfaces has been a major area of research, which traces back to questions raised by Erdős more than 60 years ago [19]. The major milestone in this area is the 1983 paper of Szemerédi and Trotter [39], proving that the maximum number of incidences between m points and n lines in the plane is $\Theta(m^{2/3}n^{2/3} + m + n)$. Since then, significant progress has been made, involving bounds on incidences with other kinds of curves or surfaces, new techniques that have simplified and extended the analysis, and related topics, such as repeated and distinct distances, and other repeated patterns. I will review the state of the art, and mention many open problems. An excellent source of many open problems in this area is the recent monograph of Brass et al. [11]. See also the monographs of Pach and Agarwal [32] and of Matoušek [28], and the survey by Pach and Sharir [33].

(d) k-Sets and levels: What is the maximum possible number of vertices in an arrangement of n lines in the plane, each having exactly k lines passing below it? This simple question is representative of many related problems, for which, in spite of almost 40 years of research, tight answers are still elusive. For example, for the question just asked, the best known upper bound is $O(nk^{1/3})$ [16], and the best known lower bound is $\Omega(n \cdot 2^{c\sqrt{\log k}})$ [30, 41]. Beyond the challenge of tightening these bounds, the same question can be asked for arrangements of hyperplanes in any dimension $d \geq 3$, where the known upper and lower bounds are even wider apart [28, 29, 36], and for arrangements of curves in the plane, where several weaker (but subquadratic) bounds have recetly been established (see, e.g., [12]). I will mention a few of the known results and the implied chal-

lenges. Good sources on these problems are Matoušek [28] and a recent survey by Wagner [42].

(e) Generalized Voronoi diagrams: Given a collection S of n *sites* in $\mathbb{R}^d$, and a metric ρ, the Voronoi diagram $Vor_\rho(S)$ is a decomposition of $\mathbb{R}^d$ into cells, one per site, so that the cell of site s consists of all the points for which s is their ρ-nearest neighbor in S. This is one of the most basic constructs in computational geometry, and yet, already in three dimensions, very few sharp bounds are known for the combinatorial complexity of Voronoi diagrams. In three dimensions, the main conjecture is that, under reasonable assumptions concerning the shape of the sites and the metric ρ, the diagram has nearly quadratic complexity. This is a classical result (with tight worst-case quadratic bound) for point sites and the Euclidean metric, but proving nearly quadratic bounds in any more general scenario becomes an extremely hard task, and only very few results are known; see [10, 14, 24, 26]. I will mention the known results and the main challenges. One of my favorites concerns *dynamic Voronoi diagrams* in the plane: If S is a set of n points, each moving at some fixed speed along some line, what is the maximum number of topological changes in the dynamically varying Voronoi diagram of S? The goal is to tighten the gap between the known nearly-cubic upper bound and nearly-quadratic lower bound. See [9, 35] for more details.

(f) Applications to range searching, optimization, and visibility: Arrangements are a fascinating structure to explore for its own sake, but they do have a myriad of applications in diverse areas. As a matter of fact, much of the study of the basic theory of arrangements has been motivated by questions arising in specific applications. I will (attempt to) highlight a few of those applications, and discuss some of the open problems that they still raise.

References

[1] Agarwal, P.K., Gao, J., Guibas, L., Koltun, V., Sharir, M.: Stable Delaunay graphs(unpublished manuscript 2007)
[2] Agarwal, P.K., Nevo, E., Pach, J., Pinchasi, R., Sharir, M., Smorodinsky, S.: Lenses in arrangements of pseudocircles and their applications. J. ACM 51, 13–186 (2004)
[3] Agarwal, P.K., Pach, J., Sharir, M.: State of the union, of geometric objects: A review. In: Proc. Joint Summer Research Conference on Discrete and Computational Geometry—Twenty Years later, Contemp. Math, AMS, Providence, RI (to appear)
[4] Agarwal, P.K., Sharir, M.: Efficient algorithms for geometric optimization. ACM Computing Surveys 30, 412–458 (1998)
[5] Agarwal, P.K., Sharir, M.: Davenport-Schinzel sequences and their geometric applications. In: Sack, J.R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 1–47, North-Holland, (2000)
[6] Agarwal, P.K., Sharir, M.: Arrangements of surfaces in higher dimensions. In: Sack, J.R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 49–119, North-Holland, (2000)

[7] Agarwal, P.K., Sharir, M.: Pseudoline arrangements: Duality, algorithms and applications. SIAM J. Comput. 34, 526–552 (2005)
[8] Aronov, B., Sharir, M.: Cutting circles into pseudo-segments and improved bounds for incidences. Discrete Comput. Geom. 28, 475–490 (2002)
[9] Aurenhammer, F., Klein, R.: Voronoi diagrams. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 201–290. Elsevier Science, Amsterdam (2000)
[10] Boissonnat, J.-D., Sharir, M., Tagansky, B., Yvinec, M.: Voronoi diagrams in higher dimensions under certain polyhedral distance functions. Discrete Comput. Geom. 19, 485–519 (1998)
[11] Braß, P., Moser, W., Pach, J.: Research Problems in Discrete Geometry. Springer, New York (2005)
[12] Chan, T.M.: On levels in arrangements of curves. Discrete Comput. Geom. 29, 375–393 (2003)
[13] Chazelle, B., Edelsbrunner, H., Guibas, L., Sharir, M.: A singly exponential stratification scheme for real semi–algebraic varieties and its applications. Theoretical Computer Science 84, 77–105 (1991) (Also in Proc. 16th Int. Colloq. on Automata, Languages and Programming, pp. 179–193 (1989))
[14] Chew, L.P., Kedem, K., Sharir, M., Tagansky, B., Welzl, E.: Voronoi diagrams of lines in three dimensions under polyhedral convex distance functions. J. Algorithms 29, 238–255 (1998)
[15] Clarkson, K., Edelsbrunner, H., Guibas, L., Sharir, M., Welzl, E.: Combinatorial complexity bounds for arrangements of curves and spheres. Discrete Comput. Geom. 5, 99–160 (1990)
[16] Dey, T.K.: Improved bounds for planar k-sets and related problems. Discrete Comput. Geom. 19, 373–382 (1998)
[17] Edelsbrunner, H.: Algorithms in Combinatorial Geometry. Springer, Berlin (1987)
[18] Edelsbrunner, H., Seidel, R.: Voronoi diagrams and arrangements. Discrete Comput. Geom. 1, 25–44 (1986)
[19] Erdös, P.: On sets of distances of n points. American Mathematical Monthly 53, 248–250 (1946)
[20] Erdös, P., Lovász, L., Simmons, A., Straus, E.G.: Dissection graphs of planar point sets. In: Srivastava, J.N., et al. (eds.) A Survey of Combinatorial Theory, pp. 139–149. North-Holland, Amsterdam (1973)
[21] Ezra, E., Sharir, M.: Almost tight bound for the union of fat tetrahedra in three dimensions. In: Proc. 48th Annu. IEEE Sympos. Foundat. Vomput. Sci (to appear, 2007)
[22] Kedem, K., Livne, R., Pach, J., Sharir, M.: On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. Discrete Comput. Geom. 1, 59–71 (1986)
[23] Koltun, V.: Almost tight upper bounds for vertical decomposition in four dimensions. J. ACM 51, 699–730 (2004)
[24] Koltun, V., Sharir, M.: Three-dimensional Euclidean Voronoi diagrams of lines with a fixed number of orientations. SIAM J. Comput. 32, 616–642 (2003)
[25] Koltun, V., Sharir, M.: The partition technique for the overlay of envelopes. SIAM J. Comput. 32, 841–863 (2003)
[26] Koltun, V., Sharir, M.: Polyhedral Voronoi diagrams of polyhedra in three dimensions. Discrete Comput. Geom. 31, 83–124 (2004)
[27] Lovász, L.: On the number of halving lines. Ann. Univ. Sci. Budapest, Eötvös, Sec. Math. 14, 107–108 (1971)

[28] Matoušek, J.: Lectures on Discrete Geometry. Springer, Heidelberg (2002)
[29] Matoušek, J., Sharir, M., Smorodinsky, S., Wagner, U.: On k-sets in four dimensions. Discrete Comput. Geom. 35, 177–191 (2006)
[30] Nivasch, G.: An improved, simple construction of many halving edges. In: Proc. Joint Summer Research Conference on Discrete and Computational Geometry—Twenty Years later, Contemp. Math, AMS, Providence, RI (to appear)
[31] Pach, J.: Finite point configurations. In: O'Rourke, J., Goodman, J. (eds.) Handbook of Discrete and Computational Geometry, pp. 3–18. CRC Press, Boca Raton, FL (1997)
[32] Pach, J., Agarwal, P.K.: Combinatorial Geometry. Wiley, New York (1995)
[33] Pach, J., Sharir, M.: Geometric incidences. In: Pach, J. (ed.) Towards a Theory of Geometric Graphs Contemporary Mathematics, Amer. Math. Soc., Providence, RI, vol. 342, pp. 185–223 (2004)
[34] Sharir, M.: Arrangements of surfaces in higher dimensions, in Advances in Discrete and Computational Geometry. In: Chazelle, B., Goodman, J.E., Pollack, R. (eds.) Proc. 1996 AMS Mt. Holyoke Summer Research Conference, Contemporary Mathematics No. 223, American Mathematical Society, pp. 335–353 (1999)
[35] Sharir, M., Agarwal, P.K.: Davenport-Schinzel Sequences and Their Geometric Applications. Cambridge University Press, Cambridge (1995)
[36] Sharir, M., Smorodinsky, S., Tardos, G.: An improved bound for k-sets in three dimensions. Discrete Comput. Geom. 26, 195–204 (2001)
[37] Steiner, J.: Einige Gesetze über die Theilung der Ebene und des Raumes, J. Reine Angew. Math. 1, 349–364 (1826)
[38] Székely, L.: Crossing numbers and hard Erdös problems in discrete geometry. Combinatorics, Probability and Computing 6, 353–358 (1997)
[39] Szemerédi, E., Trotter, W.T.: Extremal problems in discrete geometry. Combinatorica 3, 381–392 (1983)
[40] Tamaki, H., Tokuyama, T.: How to cut pseudo-parabolas into segments. Discrete Comput. Geom. 19, 265–290 (1998)
[41] Tóth, G.: Point sets with many k-sets. Discrete Comput. Geom. 26, 187–194 (2001)
[42] Wagner, U.: k-Sets and k-facets, manuscript (2006)

Nash Equilibria in Voronoi Games on Graphs*

Christoph Dürr and Nguyen Kim Thang

LIX, CNRS UMR 7161, Ecole Polytechnique 91128 Palaiseau, France
{durr,thang}@lix.polytechnique.fr

Abstract. In this paper we study a game where every player is to choose a vertex (facility) in a given undirected graph. All vertices (customers) are then assigned to closest facilities and a player's payoff is the number of customers assigned to it. We show that deciding the existence of a Nash equilibrium for a given graph is $\mathcal{NP}$-hard. We also introduce a new measure, the *social cost discrepancy*, defined as the ratio of the costs between the worst and the best Nash equilibria. We show that the social cost discrepancy in our game is $\Omega(\sqrt{n/k})$ and $O(\sqrt{kn})$, where n is the number of vertices and k the number of players.

1 Introduction

Voronoi game is a widely studied game which plays on a continuous space, typically a 2-dimensional rectangle. Players alternatively place points in the space. Then the Voronoi diagram is considered. Every player gains the total surface of the Voronoi cells of his points [1]. This game is related to the facility location problem, where the goal is to choose a set of k facilities in a bipartite graph, so to minimize the sum of serving cost and facility opening cost [8].

We consider the discrete version of the Voronoi game which plays on a given graph instead on a continuous space. Whereas most papers about these games [3,5] study the existence of a winning strategy, or computing the best strategy for a player, we study in this paper the Nash equilibria.

Formally the discrete Voronoi game plays on a given undirected graph $G(V, E)$ with $n = |V|$ and k players. Every player has to choose a vertex (*facility*) from V, and every vertex (*customer*) is assigned to the closest facilities. A player's payoff is the number of vertices assigned to his facility. We define the *social cost* as the sum of the distances to the closest facility over all vertices.

We consider a few typical questions about Nash Equilibria:

- Do Nash equilibria exist?
- What is the computational complexity for finding one?
- If they exist, can one be found from an arbitrary intital strategy profile with the best-response dynamic?
- It they exist, how different are their social costs?

* Supported by ANR Alpage.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 17–28, 2007.

The existence of Nash equilibria is a graph property for a fixed number of players, and we give examples of graphs for which there exist Nash equilibria and examples for which there are none. We show that deciding this graph property is an $\mathcal{NP}$-hard problem.

For particular graphs, namely the cycles, we characterize all Nash equilibria. We show that the best-response dynamic does not converge to a Nash equilibria on these graphs, but does for a modified version of this game.

Finally we introduce a new measure. Assume that there are Nash equilibria for k players on graph G. Let A be the largest social cost of a Nash equilibrium, B the smallest social cost of a Nash equilibrium, and C the smallest social cost of any strategy profile, which is not necessarily an equilibrium. Then the ratio A/C is called the price of anarchy [7], B/C is called the price of stability [2]. We study a different ratio A/B, which we call the *social cost discrepancy* of G. The social cost discrepancy of the game is defined as the worst discrepancy over all instances of the game. The idea is that a small social cost discrepancy guarantees that the social costs of Nash equilibria do not differ too much, and measures a degree of choice in the game. Moreover, in some settings it may be unfair to compare the cost of a Nash equilibrium with the optimal cost, which may not be attained by selfish agents. Note that this ratio is upper-bounded by the price of anarchy. We show that the social cost discrepancy in our game is $\Omega(\sqrt{n/k})$ and $O(\sqrt{kn})$. Hence for a constant number of players we have tight bounds.

2 The Game

For this game we need to generalize the notion of vertex partition of a graph: A *generalized partition* of a graph $G(V, E)$ is a set of n-dimensional non-negative vectors, which sum up to the vector with 1 in every component, for $n = |V|$.

The Voronoi game on graphs consists of:

- A graph $G(V, E)$ and k players. We assume $k < n$ for $n = |V|$, otherwise the game has a trivial structure. The graph induces a distance between vertices $d : V \times V \to \mathbb{N} \cup \{\infty\}$, which is defined as the minimal number of edges of any connecting path, or infinite if the vertices are disconnected.
- The strategy set of each player is V. A strategy profile of k players is a vector $f = (f_1, \ldots, f_k)$ associating each player to a vertex.
- For every vertex $v \in V$ — called *customer* — the distance to the closest facility is denoted as $d(v, f) := \min_{f_i} d(v, f_i)$. Customers are assigned in equal fractions to the closest facilities as follows. The strategy profile f defines the generalized partition $\{F_1, \ldots, F_k\}$, where for every player $1 \leq i \leq k$ and every vertex $v \in V$,

$$F_{i,v} = \begin{cases} \frac{1}{|\arg\min_j d(v,f_j)|} & \text{if } d(v, f_i) = d(v, f) \\ 0 & \text{otherwise.} \end{cases}$$

  We call F_i the *Voronoi cell* of player i. Now the payoff of player i is the (fractional) amount of customers assigned to it (see figure 1), that is $p_i := \sum_{v \in V} F_{i,v}$.

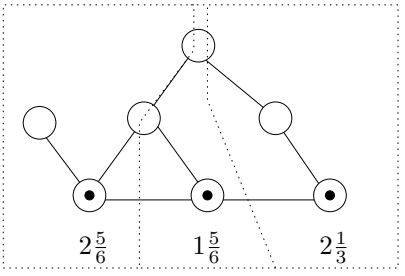


Fig. 1. A strategy profile of a graph (players are dots) and the corresponding payoffs

The *best response* for player i in the strategy profile f is a vertex $v \in V$ maximizing the player i's payoff in the strategy profile which equals f except that the strategy of player i is v. If player i can strictly improve his payoff by choosing another strategy, we say that player i is *unhappy* in f, otherwise he is *happy*. The *best response dynamic* is the process of repeatedly choosing an arbitrary unhappy player, and change it to an arbitrary best response. A *pure Nash equilibrium* is defined as a fixed point to the best response dynamic, or equivalently as a strategy profile where all players are happy. In this paper we consider only pure Nash equilibria, so we omit from now on the adjective "pure".

We defined players' payoffs in such a way, that there is a subtle difference between the Voronoi game played on graphs and the Voronoi game played on a continuous surface. Consider a situation where a player i moves to a location already occupied by a single player j, then in the continuous case player i gains exactly a half of the previous payoff of player j (since it is now shared with i). However, in our setting (the discrete case), player i can sometimes gain more than a half of the previous payoff of player j (see figure 2).

Also note that the best responses for a player in our game are computable in polynomial time, whereas for the Voronoi game in continuous space, the problem seems hard [4].

A simple observation leads to the following bound on the players payoff.

Lemma 1. *In a Nash equilibrium the payoff p_i of every player i is bounded by $n/2k < p_i < 2n/k$.*

Proof: If a player gains p and some other player moves to the same location then both payoffs are at least $p/2$. Therefore the ratio between the largest and the smallest payoffs among all players can be at most 2. If all players have the same payoff, it must be exactly n/k, since the payoffs sum up to n. Otherwise there is at least one player who gains strictly less than n/k, and another player who gains strictly more than n/k. This concludes the proof. □

3 Example: The Cycle Graph

Let $G(V, E)$ be the cycle on n vertices with $V = \{v_i : i \in \mathbb{Z}_n\}$ and $E = \{(v_i, v_{i+1}) : i \in \mathbb{Z}_n\}$, where addition is modulo n. The game plays on the

undirected cycle, but it will be convenient to fix an orientation. Let $u_0, \ldots, u_{\ell-1}$ be the distinct facilities chosen by k players in a strategy profile f with $\ell \leq k$, numbered according to the orientation of the cycle. For every $j \in \mathbb{Z}_\ell$, let $c_j \geq 1$ be the number of players who choose the facility u_j and let $d_j \geq 1$ be the length of the directed path from u_j to u_{j+1} following the orientation of G. Now the strategy profile is defined by these 2ℓ numbers, up to permutation of the players. We decompose the distance into $d_j = 1 + 2a_j + b_j$, for $0 \leq b_j \leq 1$, where $2a_j + b_j$ is the number of vertices between facilities u_j and u_{j+1}. So if $b_j = 1$, then there is a vertex in midway at equal distance from u_j and u_{j+1}.

With these notations the payoff of player i located on facility u_j is

$$p_i := \frac{b_{j-1}}{c_{j-1} + c_j} + \frac{a_{j-1} + 1 + a_j}{c_j} + \frac{b_j}{c_j + c_{j+1}}.$$

All Nash equilibria are explicitly characterized by the following lemma. The intuition is that the cycle is divided by the players into segments of different length, which roughly differ at most by a factor 2. The exact statement is more subtle because several players can be located at a same facility and the payoff is computed differently depending on the parity of the distances between facilities.

Lemma 2. *For a given strategy profile, let γ be the minimal payoff among all players, i.e., $\gamma := \min\{p_i | 1 \leq i \leq k\}$. Then this strategy profile is a Nash equilibrium if and only if, for all $j \in \mathbb{Z}_\ell$:*

(i) $c_j \leq 2$
(ii) $d_j \leq 2\gamma$
(iii) If $c_j = 1$ *and* $d_{j-1} = d_j = 2\gamma$ *then* $c_{j-1} = c_{j+1} = 2$.
(iv) If $c_{j-1} = 2, c_j = 1, c_{j+1} = 1$ *then* d_{j-1} *is odd.*
If $c_{j-1} = 1, c_j = 1, c_{j+1} = 2$ *then* d_j *is odd.*

The proof is purely technical and omitted.

A method to find Nash equilibrium in some games is to apply the best-response mechanism from an initial strategy profile. However, in our game, even in the simple cycle graph in which all Nash equilibria can be exactly characterized, there is no hope to use the best-response mechanism to find Nash equilibrium.

Lemma 3. *On the cycle graph, the best response dynamic does not converge.*

Proof: Figure 2 shows an example of a graph, where the best response dynamic can iterate forever. □

Nevertheless there is a slightly different Voronoi game in which the best response dynamic converges : The *Voronoi game with disjoint facilities* is identical with the previous game, except that players who are located on the same facility now gain zero.

Lemma 4. *On the cycle graph, for the* Voronoi game with disjoint facilities*, the best response dynamic does converge on a strategy profile in which players are located on distinct facilities.*

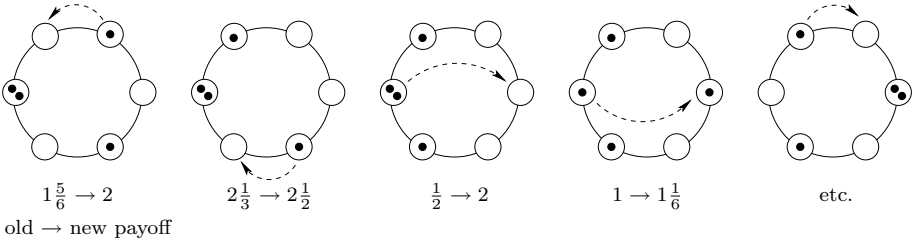


Fig. 2. The best response dynamic does not converge on this graph

Proof: To show convergence we use a potential function. For this purpose we define the *dominance order*: Let A, B be two multisets. If $|A| < |B|$ then $A \succ B$. If $|A| = |B| \geq 1$, and $\max A > \max B$ then $A \succ B$. If $|A| = |B| \geq 1, \max A = \max B$ and $A \backslash \{\max A\} \succ B \backslash \{\max B\}$ then $A \succ B$. This is a total order.

The potential function is the multiset $\{d_0, d_1, \ldots, d_{k-1}\}$, that is all distances between successive occupied facilities. Player i's payoff — renumbered conveniently — is simply $(d_i + d_{i+1})/2$. Now consider a best response for player i moving to a vertex not yet chosen by another player, say between player j and $j+1$. Therefore in the multiset $\{d_0, d_1, \ldots, d_{k-1}\}$, the values d_i, d_{i+1}, d_j are replaced by $d_i + d_{i+1}, d', d''$ for some values $d', d'' \geq 1$ such that $d_j = d' + d''$. The new potential value is dominated by the previous one. This proves that after a finite number of iterations, the best response dynamic converges to a Nash equilibrium. □

4 Existence of a Nash Equilibrium is $\mathcal{NP}$-Hard

In this section we show that it is $\mathcal{NP}$-hard to decide whether for a given graph $G(V, E)$ there is a Nash equilibrium for k players. For this purpose we define a more general but equivalent game, which simplifies the reduction.

In the *generalized Voronoi game* $\langle G(V, E), U, w, k\rangle$ we are given a graph G, a set of facilities $U \subseteq V$, a positive weight function w on vertices and a number of players k. Here the set of strategies of each player is only U instead of V. Also the payoff of a player is the weighted sum of fractions of customers assigned to it, i.e., the payoff of player i is $p_i := \sum_{v \in V} w(v) F_{i,v}$.

Lemma 5. *For every generalized Voronoi game $\langle G(V, E), U, w, k\rangle$ there is a standard Voronoi game $\langle G'(V', E'), k\rangle$ with $V \subseteq V'$, which has the same set of Nash equilibria and which is such that $|V'|$ is polynomial in $|V|$ and $\sum_{v \in V} w(v)$.*

Proof: To construct G' we will augment G in two steps. Start with $V' = V$.

First, for every vertex $u \in V$ such that $w(u) > 1$, let H_u be a set of $w(u) - 1$ new vertices. Set $V' = V' \cup H_u$ and connect u with every vertex from H_u.

Second, let H be a set of $k(a+1)$ new vertices where $a = |V'| = \sum_{v \in V} w(v)$. Set $V' = V' \cup H$ and connect every vertex of U with every vertex of H.

Now in $G'(V', E')$ every player's payoff can be decomposed in the part gained from $V'\backslash H$ and the part gained from H. We claim that in a Nash equilibrium every player chooses a vertex from U. If there is at least one player located in U, then the gain from H of any other player is 0 if located in $V'\backslash(U \cup H)$, is 1 if located in H and is at least $a+1$ if located in U. Since the total payoff from $V'\backslash H$ over all players is a, this forces all players to be located in U.

Clearly by construction, for any strategy profile $f \in U^k$, the payoffs are the same for the generalized Voronoi game in G as for the standard Voronoi game in G'. Therefore we have equivalence of the set of Nash equilibria in both games. □

Our $\mathcal{NP}$-hardness proof will need the following gadget.

Lemma 6. *For the graph G shown in figure 3 and $k = 2$ players, there is no Nash equilibrium.*

Proof: We will simply show that given an arbitrary location of one player, the other player can move to a location where he gains at least 5. Since the total payoff over both players is 9, this will prove that there is no Nash equilibrium, since the best response dynamic does not converge.

By symmetry without loss of generality the first player is located at the vertices u_1 or u_2. Now if the second player is located at u_6, his payoff is at least 5. □

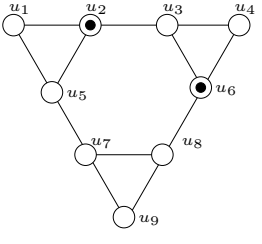


Fig. 3. Example of a graph with no Nash equilibrium for 2 players

Theorem 1. *Given a graph $G(V, E)$ and a set of k players, deciding the existence of Nash equilibrium for k players on G is $\mathcal{NP}$-complete for arbitrary k, and polynomial for constant k.*

Proof: The problem is clearly in $\mathcal{NP}$, since best responses can be computed in polynomial time, therefore it can be verified efficiently if a strategy profile is a Nash equilibrium. Since there are n^k different strategy profiles, for $n = |V|$, the problem is polynomial when k is constant.

For the proof of $\mathcal{NP}$-hardness, we will reduce 3-Partition — which is unary $\mathcal{NP}$-complete [6] — to the generalized Voronoi game, which by Lemma 5 is itself reduced to the original Voronoi game. In the 3-Partition problem we are given integers $a_1, \ldots, a_{3m}$ and B such that $B/4 < a_i < B/2$ for every

$1 \leq i \leq 3m$, $\sum_{i=1}^{3m} = mB$ and have to partition them into disjoint sets $P_1, \ldots, P_m \subseteq \{1, \ldots, 3m\}$ such that for every $1 \leq j \leq m$ we have $\sum_{i \in P_j} a_i = B$.

We construct a weighted graph $G(V, E)$ with the weight function $w : V \to \mathbb{N}$ and a set $U \subseteq V$ such that for $k = m + 1$ players ($m \geq 2$) there is a Nash equilibrium to the generalized Voronoi game $\langle G, U, w, k \rangle$ if and only if there is a solution to the 3-PARTITION instance. We define the constants $c = \binom{3m}{3} + 1$ and $d = \left\lfloor \frac{Bc - c + c/m}{5} \right\rfloor + 1$. The graph G consists of 3 parts. In the first part V_1, there is for every $1 \leq i \leq 3m$ a vertex v_i of weight $a_i c$. There is also an additional vertex v_0 of weight 1. In the second part V_2, there is for every triplet (i, j, k) with $1 \leq i < j < k \leq 3m$ a vertex u_{ijk} of unit weight. — Ideally we would like to give it weight zero, but there seems to be no simple generalization of the game which allows zero weights, while preserving the set of Nash equilibria. — Every vertex u_{ijk} is connected to v_0, v_i, v_j and v_k. The third part V_3, consists of the 9 vertex graph of figure 3 where each of the vertices $u_1, \ldots, u_9$ has weight d. To complete our construction, we define the facility set $U := V_2 \cup V_3$. Note that although the graph for the generalized Voronoi game is disconnected, the reduction of Lemma 5 to the original Voronoi game will connect V_2 with V_3.

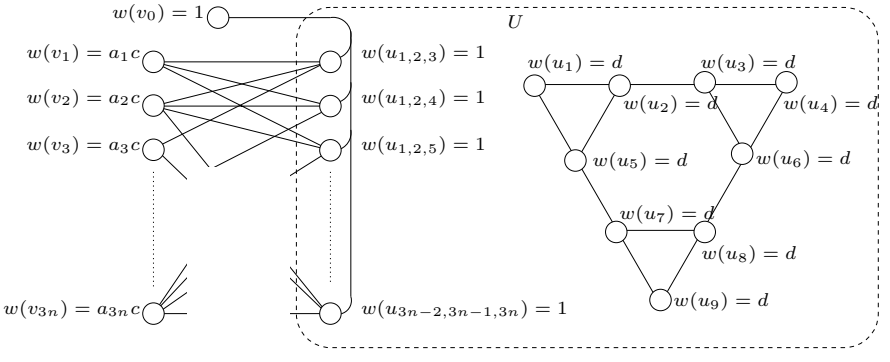


Fig. 4. Reduction from 3-PARTITION

First we show that if there is a solution $P_1, \ldots, P_m$ to the 3-PARTITION instance then there is a Nash equilibrium for this graph. Simply for every $1 \leq q \leq m$ if $P_q = \{i, j, k\}$ then player q is assigned to the vertex u_{ijk}. Player $m+1$ is assigned to u_2. Now player $(m+1)$'s payoff is $9d$, and the payoff of each other player q is $Bc + c/m$. To show that this is a Nash equilibrium we need to show that no player can increase his payoff. There are different cases. If player $m + 1$ moves to a vertex u_{ijk}, his payoff will be at most $\frac{3}{4}Bc + c/(m + 1) < 9d$, no matter if that vertex was already chosen by another player or not. If player $1 \leq q \leq m$ moves from vertex u_{ijk} to a vertex u_ℓ then his gain can be at most $5d < Bc + c/m$. But what can be his gain, if he moves to another vertex $u_{i'j'k'}$? In case where $i = i', j = j', k \neq k'$, $a_i c + a_j c$ is smaller than $\frac{3}{4}Bc$ because $a_i + a_j + a_k = B$ and $a_k > B/4$. Since $a_{k'} < B/2$, and player q gains only half of

it, his payoff is at most $a_i c + a_j c + a_{k'} c/2 + c/m < Bc + c/m$ so he again cannot improve his payoff. The other cases are similar.

Now we show that if there is a Nash equilibrium, then it corresponds to a solution of the 3-PARTITION instance. So let there be a Nash equilibrium. First we claim that there is exactly one player in V_3. Clearly if there are 2 players, this contradicts equilibrium by Lemma 6. If there are 3 players or more, then by a counting argument there are vertices v_i, v_j, v_k which are at distance more than one from any player. One of the players located at V_3 gains at most $3d$ and if he moves to u_{ijk}, his payoff would be at least $\frac{3}{4}Bc + c/m > 3d$. Now if there is no player in V_3, then any player moving to u_2 will gain $9d > \frac{3}{2}Bc + c/m$ which is an upper bound for the payoff of players located in V_2. So we know that there is a single player in V_3 and the m players in V_2 must form a partition, since otherwise there is a vertex $v_i \in V_1$ at distance at least 2 to any player. So, by the previous argument, there would be a player in V_2 who can increase his payoff by moving to the other vertex in V_2 as well. (He moves in such a way that his new facility is at distance 1 to v_i.) Moreover, in this partition, each player gains exactly $Bc + c/m$ because if one gains less, given all weights in V_1 are multiple of c, he gains at most $Bc - c + c/m$ and he can always augment his payoff by moving to V_3 ($5d > Bc - c + c/m$). □

5 Social Cost Discrepancy

In this section, we study how much the social cost of Nash equilibria can differ for a given graph, assuming Nash equilibria exist. We define the *social cost* of a strategy profile f as $\text{cost}(f) := \sum_{v \in V} d(v, f)$. Since we assumed $k < n$ the cost is always non-zero. The *social cost discrepancy* of the game is the maximal fraction $\text{cost}(f)/\text{cost}(f')$ over all Nash equilibria f, f'. For unconnected graphs, the social cost can be infinite, and so can be the social cost discrepancy. Therefore in this section we consider only connected graphs.

Lemma 7. *There are connected graphs for which the social cost discrepancy is $\Omega(\sqrt{n/k})$, where n is the number of vertices and k the number of players.*

Proof: We construct a graph G as shown in figure 5. The total number of vertices in the graph is $n = k(2a + b + 2)$. We distinguish two strategy profiles f and f': the vertices occupied by f are marked with round dots, and the vertices of f' are marked with triangular dots.

By straightforward verification, it can be checked that both f and f' are Nash equilibria. However the social cost of f is $\Theta(kb + ka^2)$ while the social cost of f' is $\Theta(kab + ka^2)$. The ratio between both costs is $\Theta(a) = \Theta(\sqrt{n/k})$ when $b = a^2$ and thus the cost discrepancy is lower bounded by this quantity. □

The *radius* of the Voronoi cell of player i is defined as $\max_v d(v, f_i)$ where the maximum is taken over all vertices v such that $F_{i,v} > 0$. The *Delaunay triangulation* is a graph H_f on the k players. There is an edge (i, j) in H_f either if there is a vertex v in G with $F_{i,v} > 0$ and $F_{j,v} > 0$ or if there is an edge (v, v') in G with $F_{i,v} > 0$ and $F_{j,v'} > 0$.

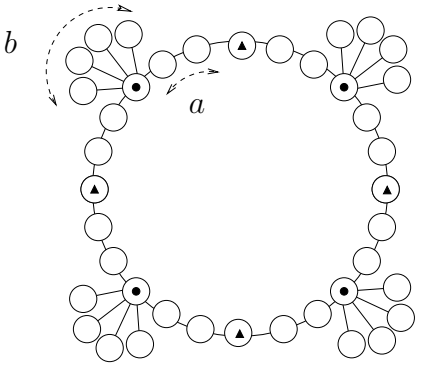


Fig. 5. Example of a graph with high social cost discrepancy

We will need to partition the Delaunay triangulation into small sets, which are 1-dominated and contain more than one vertex. We call these sets *stars*: For a given graph $G(V, E)$ a vertex set $A \subseteq V$ is a *star* if $|A| \geq 2$, and there is a *center* vertex $v_0 \in A$ such that for every $v \in A, v \neq v_0$ we have $(v_0, v) \in E$. Note that our definition allows the existence of additional edges between vertices from A.

Lemma 8. *For any connected graph $G(V, E)$, V can be partitioned into stars.*

Proof: We define an algorithm to partition V into stars.

As long as the graph contains edges, we do the following. We start choosing an edge: If there is a vertex u with a unique neighbor v, then we choose the edge (u, v); otherwise we choose an arbitrary edge (u, v). Consider the vertex set consisting of u, v as well as of any vertex w that would be isolated when removing edge (u, v). Add this set to the partition, remove it as well as adjacent edges from G and continue.

Clearly the set produced in every iteration is a star. Also when removing this set from G, the resulting graph does not contain an isolated vertex. This property is an invariant of this algorithm, and proves that it ends with a partition of G into stars. □

Note that, when a graph is partitioned into stars, the centers of these stars form a dominating set of this graph. Nevertheless, vertices in a dominating set are not necessarily centers of any star-partition of a given graph.

The following lemma states that given two different Nash equilibria f and f', every player in f is not too far from some player in f'. For this purpose we partition the Delaunay triangulation H_f into stars, and bound the distance from any player of a star to f' by some value depending on the star.

Lemma 9. *Let f be an equilibrium and A be a star of a star partition of the Delaunay triangulation H_f. Let r be the maximal radius of the Voronoi cells over all players $i \in A$. Then, for any equilibrium f', there exists a player f'_j such that $d(f_i, f'_j) \leq 6r$ for every $i \in A$.*

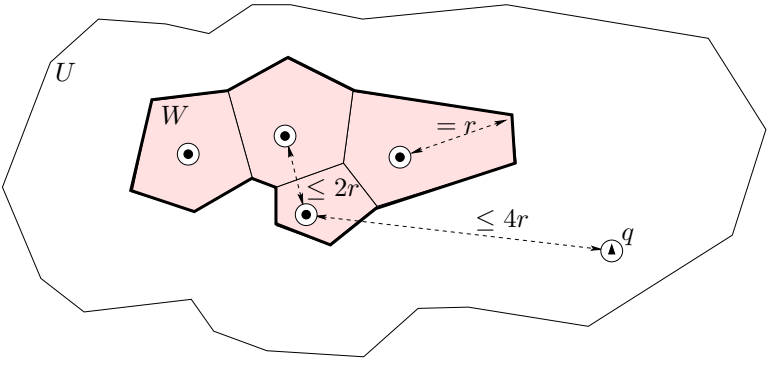


Fig. 6. Illustration of lemma 9

Proof: Let $U = \{v \in V : \min_{i \in A} d(v, f_i) \leq 4r\}$. If we can show that there is a facility $f'_j \in U$ we would be done, since by definition of U there would be a player $i \in A$ such that $d(f_i, f'_j) \leq 4r$ and the distance between any pair of facilities of A is at most $2r$. This would conclude the lemma.

So for a proof by contradiction, assume that in the strategy profile f' there is no player located in U. Now consider the player with smallest payoff in f'. His payoff is not more than n/k. However if this player would choose as a facility the center of the star A, then he would gain strictly more: By the choice of r, any vertex in W is at distance at most $3r$ to the center of the star. However, by assumption and definition of U, any other facility of f' would be at distance strictly greater than $3r$ to any vertex in W. So the player would gain at least all vertices around it at distance at most $3r$, which includes W. Since any player's payoff is strictly more than $n/2k$ by Lemma 11, and since a star contains at least two facilities by definition, this player would gain strictly more than n/k, contradicting that f' is an equilibrium. This concludes the proof. □

Theorem 2. *For any connected graph $G(V, E)$ and any number of players k the social cost discrepancy is $O(\sqrt{kn})$, where $n = |V|$.*

Proof: Let f, f' be arbitrary equilibria on $G(V, E)$. We will consider a generalized partition of V and for each part bound the cost of f' by $c\sqrt{kn}$ times the cost of f for some constant c.

For a non-negative n-dimensional vector W we define the cost restricted to W as $\text{cost}_W(f) = \sum_{v \in V} W_v \cdot d(v, f)$. Now the cost of f would write as the sum of $\text{cost}_W(f)$ over the vectors W from some fixed generalized partition.

Fix a star partition of the Delaunay triangulation H_f. Let A be an arbitrary star from this partition, $a = |A|$, and let W be the *sum* of the corresponding Voronoi cells, i.e., $W = \sum_{i \in A} F_i$. We will show that $\text{cost}_W(f') = O(\sqrt{kn} \cdot \text{cost}_W(f))$, which would conclude the proof. There will be two cases $k \leq n/4$ and $k > n/4$.

By the previous lemma there is a vertex f'_j such that $d(f_i, f'_j) \leq 6r$ for all $i \in A$, where r is the largest radius of all Voronoi cells corresponding to the star A. So the cost of f' restricted to the vector W is

$$\begin{aligned}\text{cost}_W(f') &= \sum_{v\in V} W_v \cdot d(v, f') \leq \sum_{v\in V} W_v \cdot d(v, f'_j) \\ &= \sum_{v\in V}\sum_{i\in A} F_{i,v} \cdot d(v, f'_j) \\ &\leq \sum_{v\in V}\sum_{i\in A} F_{i,v} \cdot \big(d(v, f_i) + d(f_i, f'_j)\big) \\ &\leq \text{cost}_W(f) + 6r \cdot |W|, \end{aligned} \tag{1}$$

where $|W| := \sum_{v\in V} W_v$.

Moreover by definition of the radius, there is a vertex v with $W_v > 0$ such that the shortest path to the closest facility in A has length r. So the cost of f restricted to W is bigger than the cost restricted to this shortest path:

$$\text{cost}_W(f) \geq (\frac{1}{k} \cdot 1 + \frac{1}{k} \cdot 2 + \ldots + \frac{1}{k} \cdot r) \geq \frac{1}{k} \cdot r(r-1)/2.$$

(The fraction $\frac{1}{k}$ appears because a vertex can be assigned to at most k players.)

First we consider the case $k \leq n/4$. We have

$$\text{cost}_W(f) \geq |W| - |A| \geq a(n/2k - 1) \geq an/4k.$$

The first inequality is because the distance of all customers which are not facilites to a facility is at least one. The second inequality is due to Lemma 1 and $|W|$ is the sum of payoffs of all players in A.

Note that $|W| \leq n$ and $2 \leq a \leq k$. Now if $r \leq \sqrt{an}$, then

$$\frac{\text{cost}_W(f')}{\text{cost}_W(f)} \leq 1 + \frac{6r|W|}{\text{cost}_W(f)} \leq 1 + \frac{6r \cdot a \cdot 2n/k}{an/4k} = O(r) = O(\sqrt{kn}).$$

And if $r \geq \sqrt{an}$, then

$$\frac{\text{cost}_W(f')}{\text{cost}_W(f)} \leq 1 + \frac{6r|W|}{\text{cost}_W(f)}) \leq 1 + \frac{6r \cdot a \cdot 2n/k}{r(r-1)/2k} = O(an/r) = O(\sqrt{kn}).$$

Now we consider case $k > n/4$. In any equilibrium, the maximum payoff is at most $2n/k$. Moreover the radius r of any Voronoi cell is upper bounded by $n/k + 1$, otherwise the player with minimum gain (which is at most n/k) could increase his gain by moving to a vertex which is at distance at least r from every other facility. Therefore $r = O(1)$. Summing (1) over all stars with associated partition W, we obtain $\text{cost}(f') \leq \text{cost}(f) + cn$, for some constant c. Remark that the social cost of any equilibrium is at least $n - k$. Hence, $\frac{\text{cost}(f')}{\text{cost}(f)} = O(n)$. □

6 Open Problems

It would be interesting to close the gap between the lower and upper bounds for the social cost discrepancy. The price of anarchy is still to be studied. Just notice that it can be as large as $\Omega(n)$, as for the star graph and $k = n - 1$ players: The unique Nash equilibria locates all players in the center, while the optimum is to place every player on a distinct leaf. Furthermore, it is expected that the social cost discrepancy would be considered in other games in order to better understand Nash equilibria in these games.

References

1. Ahn, H.-K., Cheng, S.-W., Cheong, O., Golin, M., van Oostrum, R.: Competitive facility location: the Voronoi game. Theoretical Computer Science 310, 457–467 (2004)
2. Anshelevich, E., Dasgupta, A., Kleinberg, J., Tardos, E., Wexler, T., Roughgarden, T.: The price of stability for network design with fair cost allocation. In: FOCS '04, pp. 295–304 (2004)
3. Cheong, O., Har-Peled, S., Linial, N., Matousek, J.: The One-Round Voronoi Game. Discrete Comput. Geom. 31(1), 125–138 (2004)
4. Dehne, F., Klein, R., Seidel, R.: Maximizing a Voronoi region: the convex case. International Journal of Computational Geometry 15(5), 463–475 (2005)
5. Fekete, S.P., Meijer, H.: The one-round Voronoi game replayed. Computational Geometry: Theory and Applications 30, 81–94 (2005)
6. Garey, M.R., Johnson, D.S.: Complexity results for multiprocessor scheduling under resource constraints. SIAM Journal on Computing 4, 397–411 (1975)
7. Koutsoupias, E., Papadimitriou, C.: Worst-Case Equilibria. In: Meinel, C., Tison, S. (eds.) STACS 99. LNCS, vol. 1563, pp. 404–413. Springer, Heidelberg (1999)
8. Vetta, A.: Nash equilibria in competitive societies with applications to facility location. In: Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 416–425. IEEE Computer Society Press, Los Alamitos (2002)

Evolutionary Equilibrium in Bayesian Routing Games: Specialization and Niche Formation

Petra Berenbrink* and Oliver Schulte**

School of Computing Science
Simon Fraser University

Abstract. In this paper we consider Nash Equilibria for the selfish routing model proposed in [12], where a set of n users with tasks of different size try to access m parallel links with different speeds. In this model, a player can use a mixed strategy (where he uses different links with a positive probability); then he is indifferent between the different link choices. This means that the player may well deviate to a different strategy over time. We propose the concept of evolutionary stable strategies (ESS) as a criterion for stable Nash Equilibria, i.e. Equilibria where no player is likely to deviate from his strategy. An ESS is a steady state that can be reached by a user community via evolutionary processes in which more successful strategies spread over time. The concept has been used widely in biology and economics to analyze the dynamics of strategic interactions. We establish that the ESS is *uniquely determined* for a symmetric Bayesian parallel links game (when it exists). Thus evolutionary stability places strong constraints on the assignment of tasks to links.

Keywords: Selfish Routing, Bayesian Games, Evolutionarily Stable Strategy.

1 Introduction

We consider the *selfish routing model* proposed in [12], where users try to access a set of parallel links with different speeds. This scenario gives rise to a strategic interaction between users that combines aspects of both competition, in that users compete for the fastest links, and coordination, in that users want to avoid overloaded links. Koutsoupias and Papadimitriou suggest to study the model in a game-theoretic frame work [12]. They compare the cost of the worst case Nash equilibrium with the cost of an optimal solution; this ratio was called *price of anarchy*. Depending on the cost function that is used to assess the optimal solution, the ratio between Nash equilibria and optimal solutions can vary greatly. For example, the cost of the worst case Nash equilibrium can be

* Partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC) discovery grant.

** Partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC) discovery grant.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 29–40, 2007.

similar to the cost of the optimal solution (min-max function considered in [2]), or the cost for every Nash Equilibrium can be far away from that of the optimal solution [1].

It is an elementary fact that if a player plays a *mixed Nash strategy*, then he is indifferent between the choices that carry positive probability. So, it is not easy to see what keeps the players from deviating to a different strategy with different probabilities. A Nash Equilibrium having a sequence of single-player strategy changes that do not alter their own payoffs but finally lead to a non-equilibrium position, is called **transient** [6]. Games can have several non-stable and transient Nash Equilibria, and it is unlikely that a system will end up in one of these. Hence it might be interesting to answer first the question which Nash Equilibria are stable, and then to compare the cost of stable equilibria to the cost of the optimal solution (see [6]). Several stability models were suggested in the literature [18]. One of the most important models is Maynard Smith's concept of an *evolutionarily stable strategy*, abbreviated ESS [14]. The criterion proposed by Maynard Smith is as follows: An equilibrium E is evolutionarily stable if there is a threshold $\overline{\varepsilon}$ for the size of deviations such that if the fraction of deviating players falls below $\overline{\varepsilon}$, then the players following the equilibrium E do better than the deviants.

1.1 New Results

In this paper we study evolutionarily stable equilibria for selfish routing in Koutsoupias and Papadimitriou's *parallel links model* [12]. We first define a symmetric version of a Bayesian parallel links game where every player is not assigned a task of a fixed size but, instead, is randomly assigned a task drawn from a distribution (Section 2.1). Then we argue that every ESS in this game is a *symmetric* Nash Equilibrium, where every player uses the same strategy.

In Section 3 we show that the symmetric Nash Equilibrium is unique for *link groups*. By *link group uniqueness* we mean the following. Assume that all links with the same speed are grouped together into so-called *link groups*. Then, in every symmetric Nash Equilibrium, the total probability that tasks of a certain size are sent to a link group is unique. This implies that the only flexibility in a symmetric Nash Equilibrium is the probability distribution over links from the same link group, not over different link groups. Then we show that in a symmetric equilibrium two links with different speeds cannot both be used by two ore more tasks with different weights. In fact, we show an even stronger result: If link ℓ is used for task w and ℓ' for $w' \neq w$, then at least one of the links will not be optimal for the other link's task. We also show that tasks with larger weight must be assigned to links with bigger speed.

In Section 4 we characterize ESS for the symmetric Bayesian parallel links game. We show that every ESS is a Nash Equilibrium, and we show that, to evaluate evolutionary stability, we have to consider only best replies to the current strategy. Then we establish that in an ESS, we not only have link group uniqueness, but also the probability distribution with which links of the same group are chosen by tasks has to be unique. In fact, an ESS requires treating two

links with equal speed exactly the same. This result establishes the *uniqueness of ESS*. We show that in an ESS even two links with the same speed cannot both be used by two ore more tasks with different weights. This implies that in an ESS links acquire niches, meaning that there is minimal overlap in the tasks served by different links. We call this *specialization* in the following.

In general, the problem of calculating an ESS is very hard; it is contained in Σ_2^P and is both NP-hard and coNP-hard [5]. We expect that our uniqueness results and the structual properties of ESS for our game will help to develop algorithms that compute an ESS. We also show that, unfortunately, the specialisation condition is necessary for an ESS, but not sufficient. We introduce a sufficient condition called *clustering*—roughly, links must form disjoint niches—and show that every clustered Nash equilibrium is an ESS. Unfortunately, we also show that there exists a game that does not have a clustered ESS, but it has an unclustered ESS, so clustering is not a necessary condition.

1.2 Known Results

The Parallel Links Game was introduced by Koutsoupias and Papadimitriou [12], who initiated the study of coordination ratios. In the model of [12], the cost of a collection of strategies is the (expected) maximum load of a link (maximized over all links). The coordination ratio is defined as the ratio between the maximum cost (maximized over all Nash equilibria) divided by the cost of the optimal solution. Koutsoupias and Papadimitriou give bounds on the coordination ratio. These bounds are improved by Mavronicolas and Spirakis [13], and by Czumaj and Vöcking [3] who gave an asymptotically tight bound.

In [8] Gairing et al. introduce a Bayesian version of the selfish routing game. Following Harsanyi's approach [9], each user can have a set of possible types. Their paper presents a comprehensive collection of results for the Bayesian routing game. Note that their model is more general than ours since they allow different types for different users, whereas our users all have the same type space (possible task assignments). In [7] Fischer and Vöcking adopt an evolutionary approach to a related routing problem (see [16] for a definition).

The concept of evolutionary stability is fundamental in evolutionary game theory, which has many applications in theoretical biology and economics. The seminal presentation of the concept of an ESS is due to Maynard Smith [14]. Since then, the concept has played a central role in evolutionary biology and has been used in thousands of studies. Economists have also applied the concept to analyze many economic and social interactions. Kontogiannis and Spirakis provide an introduction to and motivation for evolutionary analysis from a computer science perspective [11, Sec.3]. Kearns and Suri examine evolutionary stability in graphical games [10].

Sandholm proposes a pricing scheme based on evolutionary stability for minimizing traffic congestion; he notes the potential applicability of his models to computer networks [17, Sec.8]. His approach does not apply the concept of evolutionarily stable strategy. To our knowledge, our combination of congestion game + Bayesian incomplete information + ESS is new in the literature. (Ely

and Sandholm remark that "nearly all work in evolutionary game theory has considered games of compete information" [4, p.84].)

2 Basic Models and Concepts

We first introduce Bayesian Parallel Links Games and show some simple observations concerning link load and utilities. We then introduce population games and define evolutionary stable strategies (ESS).

2.1 Bayesian Parallel Links Games

We examine an extension of the original routing game called *Bayesian parallel links game*. Our definition below is a special symmetric case of the definition in [8]. Harsanyi [9] introduced the notion of a Bayesian game to analyze games with *incomplete information* where players are uncertain about some aspect of the game such as what preferences or options the other players have. Bayesian games have found many applications in economics; eventually Harsanyi's work earned him the Nobel Prize. In a Bayesian parallel links game, the uncertainty among the players concerns the task size of the opponents. An agent knows the size of her own message, but not the size of the messages being sent by other users. The Bayesian game of [8] models this uncertainty by a distribution that specifies the probability that w is the task of user i. In our symmetric Bayesian routing model, this distribution is the same for all agents. A natural interpretation of this assumption is that agents are assigned tasks drawn from a common distribution.

A game is *symmetric* if (1) all players have the same set of strategy options, and (2) all that matters is what strategies are chosen, and how often they are chosen, but not by which player. Our Bayesian version of the game is symmetric, whereas the parallel links game is symmetric for uniform users only. The formal definition of a symmetric Bayesian routing model is as follows.

Definition 1. *A* ***symmetric Bayesian routing model*** *is a tuple* $\langle N, W, \mu, L, P\rangle$ *where*

1. $N = [n]$ *is the set of users*
2. $W = \{w_1, w_2, \ldots, w_k\}$ *is a finite set of task weights, and* $\mu : W \to (0, 1]$ *is a probability distribution over the weights* W*. The distribution* μ *is used to assign weights i.u.r. to players* $1, \ldots, n$*.*
3. $L = [m]$ *is the set of links. For* $i \in [m]$*, link* i *has speed* c_i*.*
4. *An allocation* s *is a vector in* $[0, 1]^m$ *such that* $\sum_{j=1}^{m} s(j) = 1$*.*
5. *A mixed strategy* p *is a total function* $W \to [0, 1]^m$*, i.e., a strategy assigns an allocation to every task weight in* W*. Then* P *is the set of all strategies. A strategy profile* $p_1, \ldots, p_n$ *assigns a strategy* $p_i \in P$ *to each player* i*.*

Now fix a routing model with strategy set P. In the following we use $p_i(j, w)$ for the probability that, in strategy $p_i \in P$, user i assigns a task with weight w to link j. As usual, (p_i, p_{-i}) denotes a strategy profile where user i follows strategy

p_i and the other players' strategies are given by the vector p_{-i} of length $n-1$. Similarly, (w_i, w_{-i}) denotes a weight vector where user i is assigned task size w_i and the other players' weights are given by the vector w_{-i} of length $n-1$. The concept of a mixed strategy in a symmetric Bayesian routing model may be interpreted as follows. Each player chooses a strategy before the game is played. Then tasks $w_1, w_2, ..., w_n$ are assigned i.u.r. to players 1 through n according to the distribution μ. Each player learns their own task but not that of the others. Next for each player i we "execute" the strategy p_i given task w_i, such that task w_i is sent to link j with probability $p_i(j, w)$. Thus, strategies have a natural interpretation as programs that take as input a task w and output a link for w or a probability distribution over links for w.

Like Koutsoupias and Papadimitriou [12], we assume that the latency of a link depends linearily on the load of a link. Thus we have the following definition of the load on a link.

Definition 2. *Let $B = \langle N, W, \mu, L, P\rangle$ be a symmetric Bayesian routing model. Let $p_1, .., p_n$ be a vector of mixed strategies.*

1. *We define*

$$load_i(p_1, .., p_n, w_1, ..., w_n) \equiv \frac{1}{c_i} \sum_{j=1}^{n} w_j \cdot p_j(i, w_j)$$

   *to be the **conditional expected load** on link i for* fixed $w_1, w_2, .., w_n$.
2. *We define*

$$load_i(p_1, .., p_n) \equiv \sum_{w_1,...,w_n \in W^n} load_i(p_1, .., p_n, w_1, ..., w_n) \cdot \prod_{j=1}^{n} \mu(w_j).$$

   *to be the **expected load** on link i.*

The next observation shows that the load function is additive in the sense that the total load on link i due to n users is just the sum of the loads due to the individual users. The proof can be found in the full version of this paper.

Observation 1. *Let $B = \langle N, W, \mu, L, P\rangle$ be a symmetric Bayesian routing model. Let $p_1, .., p_n$ be a vector of mixed strategies; Then for any user j we have*

$$load_i(p_1, .., p_n) = load_i(p_{-j}) + load_i(p_j).$$

Therefore $load_i(p_1, .., p_n) = \sum_{j=1}^{n} load_i(p_j)$.

A symmetric Bayesian routing game is a symmetric Bayesian routing model with utility functions for the players. In a symmetric game, the payoff of each player depends only on what strategies are chosen, and not on which players choose particular strategies. Hence for a fixed profile of opponents strategy p yields the same payoff no matter which player i follows strategy p. This allows us to simplify our notation and write $p_j(w)$ or $p_\ell(w)$ for the probability that strategy p uses a link j or l for a task with size w.

Definition 3. *A **symmetric Bayesian routing game** $B = \langle N, W, \mu, L, u, P\rangle$ is a routing model $\langle N, W, \mu, L, P\rangle$ with a utility function u. We then write $u(p; p_1, ..., p_{n-1})$ to denote the payoff of following strategy p when the other players' strategies are given by $p_1, .., p_{n-1}$. Then the payoff is defined as*

$$u(p; p_1, ..., p_{n-1}) = -\sum_{w\in W}\sum_{\ell\in L}(w/c_\ell + load_\ell(p_1, .., p_{n-1}))\cdot p_\ell(w)\cdot\mu(w).$$

Fix a symmetric Bayesian routing game B with n players. A strategy profile $(p_1, ..., p_n)$ is a Nash equilibrium if no player can improve their payoff unilaterally by changing their strategy p_i. To simplify notation for games in which several players follow the same strategy, we write p^k for a vector $(p, p, ..., p)$ with p repeated k times. Then the mixed strategy p is a **best reply** to p^{n-1} if for all mixed strategies p' we have $u(p; p^{n-1}) \geq u(p'; p^{n-1})$. If all players in a symmetric game follow the same strategy, then p^n is the resulting mixed strategy profile. The strategy profile p^n is a (symmetric) **Nash equilibrium** if p is a best reply to p^{n-1}. Hence, a symmetric Nash equilibrium for a symmetric Bayesian routing game with n players is a Nash equilibrium $(p, p, , .., p)$ in which each player follows the same strategy. It follows from Nash's existence proof [15] that a symmetric game, such as a symmetric Bayesian Routing Game, has a symmetric Nash equilibrium.

2.2 Population Games and Evolutionary Stability for the Parallel Links Game

We give a brief introduction to population games and evolutionary stability. A more extended introduction from a computer science point of view is provided in [11, Sec.3]. The standard population game model considers a very large **population** A of agents [19,14]. The agents play a symmetric game like our symmetric Bayesian routing game. Every agent in the population follows a strategy p *fixed before* the game is played. A *match* is a particular instance of the base game that results when we match n i.u.r. (independent and uniformly at random) chosen agents together to play the base game. Since strategies occur with a certain frequency in the population, there is a fixed probability with which a task with a given size is assigned to a link. *Hence, with a population A we can associate a mixed strategy that we denote by p_A.*

Consider now the expected payoff that an agent using strategy p receives in a match given a fixed population A. This is equivalent to playing strategy p against $n-1$ opponents whose choices are determined by *the same distribution*, the population distribution p_A. In other words, the expected payoff is given by $u(p; (p_A)^{n-1})$, namely the payoff of using strategy p when the other $n-1$ players follow mixed strategy p_A. A population is *in equilibrium* if no agent benefits from changing her strategy unilaterally given the state of the population. Formally, a population A with associated mixed strategy p_A is in equilibrium if every mixed strategy p that occurs with frequency greater than zero in the population is a best reply to $(p_A)^{n-1}$. It is easy to see that this is the case

if and only if the symmetric strategy profile $(p, p, .., p)$ is a Nash equilibrium. So *population equilibria correspond exactly to symmetric Nash equilibria*. While restricting attention to symmetric Nash equilibria may seem like an artificial restriction for non-population models, in large population models symmetric Nash equilibria characterize the natural equilibrium concept for a population.

The main idea in evolutionary game theory is Maynard Smith's concept of *stability against mutations*. Intuitively, a population is evolutionarily stable if a small group of mutants cannot invade the population. Consider a population A that encounters a group A^M of mutants. Then the mixed population is $A \cup A^M$. Suppose that in this mixed population the proportion of mutants is ε. The distribution for the mixed population is the probabilistic mixture $(1-\varepsilon)p_A + \varepsilon p_{A^M}$. We may view a mutation A^M as successful if the average payoff for invaders *in the mixed population* is at least as great as the average payoff for nonmutants in the mixed population. The expected payoff for a strategy p in the mixed population $A \cup A^M$ is given by $u(p; [(1-\varepsilon)p_A + \varepsilon p_{A^M}]^{n-1})$. So the average payoff for the non-mutants is $u(p_A; [(1-\varepsilon)p_A + \varepsilon p_{A^M}]^{n-1})$ and for the mutants it is $u(p_{A^M}; [(1-\varepsilon)p_A + \varepsilon p_{A^M}]^{n-1})$.

Definition 4 (ESS). *Let B be a symmetric Bayesian routing game with n players. A mixed strategy p^* is an **evolutionarily stable strategy** (ESS) $\iff$ there is an $\overline{\varepsilon} > 0$ such that for all $0 < \varepsilon < \overline{\varepsilon}$ and mixed strategies $p \neq p^*$ we have $u(p^*; [\varepsilon p + (1-\varepsilon)p^*]^{n-1}) > u(p; [\varepsilon p + (1-\varepsilon)p^*]^{n-1})$.*

3 Link Group Uniqueness of Symmetric Nash Equilibria

This section investigates the structure of symmetric Nash equilibria and establishes that symmetric equilibria are uniquely determined up to the distribution of tasks within link groups. A *link group* $\mathcal{L}$ in a symmetric Bayesian routing game B is a maximal set of links with the same speed, that is, $c_\ell = c_{\ell'}$ for all $\ell, \ell' \in \mathcal{L}$. Then, for any mixed strategy p, the probability that p sends task w to a link in link group $\mathcal{L}$ is given by $p_{\mathcal{L}}(w) \equiv \sum_{\ell \in \mathcal{L}} p_\ell(w)$. The main result of this section is that in any symmetric Bayesian routing game the aggregate distribution over groups of links with the same speed is *uniquely determined* for symmetric Nash equilibria. In other words, the probabilities $p_{\mathcal{L}}$ are uniquely determined in a symmetric Nash equilibrium; if p^n and $(p')^n$ are Nash equilibria in a routing game B, then for every link group $\mathcal{L}$ and every task weight w we have $p_{\mathcal{L}}(w) = p'_{\mathcal{L}}(w)$.

In the following we say that link ℓ is **optimal** for task w given p^{n-1} iff ℓ minimizes the function $w/c_\ell + (n-1)\cdot \mathrm{load}_\ell(p)$. In this case we write $w \in \mathrm{opt}_\ell(p)$. A mixed strategy p **uses** link ℓ for task w if $p_\ell(w) > 0$; we write $w \in \mathrm{support}_\ell(p)$. The next proposition asserts that a best reply p' to a strategy profile p^{n-1} uses a link for a task only if the link is optimal for the task given p^{n-1}. This is a variant of the standard characterization of Nash equilibrium according to which all pure strategies in the support of an equilibrium strategy are best replies. The proof can be done similar to the proof of the standard Nash characterization and is omitted.

Proposition 1. *Let B be a symmetric Bayesian routing game with n players, and let p be a mixed strategy. A strategy p' is a best reply to $p^{n-1} \iff$ for all tasks w, links ℓ, if $p'_\ell(w) > 0$, then ℓ is an optimal link for w given p^{n-1}. That is, $support_\ell(p') \subseteq opt_\ell(p)$.*

The next Lemma gives a clear picture of what a symmetric Nash equilibrium looks like. Intuitively, this picture is the following. (1) Tasks with bigger weights are placed on faster links. (2) Faster links have a bigger load. (3-5) For every link ℓ there is an "interval" of task weights $\{w_i, ..., w_l\}$ such that ℓ is optimal for all and only these weights. (6) Any pair of links with different speeds are optimal for at most one common task weight.

Lemma 1. *Let B be a symmetric Bayesian routing game with n players and a symmetric Nash equilibrium p^n. Fix any two links ℓ and ℓ'.*

1. *If $c_\ell > c_{\ell'}$, strategy p uses ℓ for task w, and p uses ℓ' for w', then $w \geq w'$.*
2. *If $c_\ell > c_{\ell'}$, then $load_\ell(p^n) > load_{\ell'}(p^n)$, or $load_\ell(p^n) = load_{\ell'}(p^n) = 0$. And if $c_\ell = c_{\ell'}$, then $load_\ell(p^n) = load_{\ell'}(p^n)$.*
3. *If $c_\ell > c_{\ell'}$, then there cannot exist tasks $w > w'$ with $w, w' \in support_\ell(p)$ and $w, w' \in support_{\ell'}(p)$.*
4. *If $w_1 \geq w_2 \geq w_3$ and $w_1 \in opt_\ell(p)$ and $w_3 \in opt_\ell(p)$, then $w_2 \in opt_\ell(p)$.*
5. *If $c_{\ell_1} \geq c_{\ell_2} \geq c_{\ell_3}$ and $w \in opt_{\ell_1}(p)$ and $w \in opt_{\ell_3}(p)$, then $opt_{\ell_2}(p) = \{w\}$.*
6. *If $c_\ell > c_{\ell'}$, then $|opt_\ell(p) \cap opt_{\ell'}(p)| \leq 1$.*

Proof. The proof can be found in the full version.

We note that Lemma 1 holds for Nash equilibria in general, not just symmetric ones. Specifically, let p' be a Nash equilibrium for a symmetric Bayesian routing game, and fix any player i such that $p' = (p'_i, p'_{-i})$. Then Lemma 1 holds if we replace a mixed strategy p with p'_i, and p^{n-1} with p'_{-i}, and p^n with (p'_i, p'_{-i}).

We extend our notation for links to link groups $\mathcal{L}$ such that $c_\mathcal{L}$ denotes the speed of all links in group $\mathcal{L}$. We also define

$$\text{load}_\mathcal{L}(p^n) \equiv \sum_{\ell \in \mathcal{L}} \text{load}_\ell(p^n).$$

The next theorem is the main result of this section. It states that for a user population in equilibrium (corresponding to a symmetric Nash equilibrium), the distribution of tasks to *link groups* is uniquely determined. Thus the only way in which population equilibria can differ is by how tasks are allocated within a link group. This result is the first key step for establishing the uniqueness of an ESS for a symmetric Bayesian routing game.

Theorem 1 (Link Group Uniqueness). *Let B be a symmetric Bayesian routing game with n players and two symmetric Nash equilibria p^n and $(p')^n$. Then we have $p_\mathcal{L}(w) = p'_\mathcal{L}(w)$ and $load_\mathcal{L}(p^n) = load_\mathcal{L}((p')^n)$ for all task sizes w and link groups $\mathcal{L}$ of B.*

The proof is in the full version. The next section investigates the structure of evolutionarily stable equilibria and proves that an ESS is uniquely determined when it exists.

4 Characterization of Evolutionary Stability

In this section we prove a necessary and sufficient condition for a mixed strategy p^* to be an ESS. In the following let $p_\ell(W) = \sum_{w \in W} p_\ell(w) \cdot \mu(w)$. The next proposition shows that for sufficiently small sizes of mutations, only best replies to the incumbent distribution p^* have the potential to do better than the incumbent. The proposition also implies that an ESS corresponds to a symmetric Nash equilibrium (Corollary 1).

Proposition 2. *Let B be a symmetric Bayesian routing game with n players, and let p^* be a mixed strategy. Then there is a threshold $\overline{\varepsilon}$ such that for all ε with $0 < \varepsilon < \overline{\varepsilon}$, for all mixed strategies p:*

1. *If $u(p^*; (p^*)^{n-1}) > u(p; (p^*)^{n-1})$, then $u(p^*; [\varepsilon p + (1-\varepsilon)p^*]^{n-1}) > u(p; [\varepsilon p + (1-\varepsilon)p^*]^{n-1})$.*
2. *If $u(p^*; (p^*)^{n-1}) < u(p; (p^*)^{n-1})$, then $u(p^*; [\varepsilon p + (1-\varepsilon)p^*]^{n-1}) < u(p; [\varepsilon p + (1-\varepsilon)p^*]^{n-1})$.*

Proof. The proof requires only standard techniques from evolutionary game theory [19] and is omitted for reasons of space. Intuitively, the result holds because we can choose our threshold $\overline{\varepsilon}$ small enough (as a function of B and p^*) so that any difference in the case in which the mutant and incumbent face 0 mutants outweighs the differences in their payofffs when they face one or more mutants.

Corollary 1. *Let B be a symmetric Bayesian routing game with n players, and let p^* be an ESS. Then $(p^*)^n$ is also a Nash Equilibrium.*

Proof (sketch). If p^* is not a best reply to $(p^*)^{n-1}$, then there is a mutant p such that $u(p; (p^*)^{n-1}) > u(p^*; (p^*)^{n-1})$. Proposition 2(2) then implies that p is a successful mutation no matter how low we choose the positive threshold $\overline{\varepsilon}$.

The next lemma 2 provides a necessary and sufficient condition for when a best reply is a successful mutation, which is key for our analysis of evolutionarily stable strategies in a given network game.

Lemma 2. *Let B be a symmetric Bayesian routing game with n players. Let $(p^*)^n$ be a Nash equilibrium, and consider any best reply p to $(p^*)^{n-1}$. Define $\Delta_\ell(p, p^*) = \sum_{\ell \in L} [load_\ell(p^*) - load_\ell(p)] \cdot [p^*_\ell(W) - p_\ell(W)]$. Then for all ε with $0 < \varepsilon < 1$, if*

1. *$\Delta_\ell(p, p^*) < 0$, then $u(p^*; [\varepsilon p + (1-\varepsilon)p^*]^{n-1}) < u(p; [\varepsilon p + (1-\varepsilon)p^*]^{n-1})$.*
2. *$\Delta_\ell(p, p^*) = 0$, then $u(p^*; [\varepsilon p + (1-\varepsilon)p^*]^{n-1}) = u(p; [\varepsilon p + (1-\varepsilon)p^*]^{n-1})$.*
3. *$\Delta_\ell(p, p^*) > 0$, then $u(p^*; [\varepsilon p + (1-\varepsilon)p^*]^{n-1}) > u(p; [\varepsilon p + (1-\varepsilon)p^*]^{n-1})$.*

The proof of Lemma 2 can be found in the full version. It shows that a best reply p to $(p^*)^{n-1}$ that has a negative quantity $\sum_{\ell \in L} [\text{load}_\ell(p^*) - \text{load}_\ell(p)] \cdot [p^*_\ell(W) - p_\ell(W)]$ is successful in the strong sense that it yields a better payoff than the incumbent strategy p^* *no matter what its size*. It will be convenient to say that a mutation p **defeats** the incumbent strategy p^* if $\sum_{\ell \in L} [\text{load}_\ell(p^*) - \text{load}_\ell(p)] \cdot$

$[p^*_\ell(W) - p_\ell(W)] < 0$. Similarly, we say that a mutation p **equals** an incumbent p^* if $\sum_{\ell \in L}[\text{load}_\ell(p^*) - \text{load}_\ell(p)] \cdot [p^*_\ell(W) - p_\ell(W)] = 0$. In this terminology our results so far yield the following characterization of evolutionary stability. The proof directly follows from Proposition 2 and Lemma 2.

Corollary 2. *Let B be a symmetric Bayesian routing game with n players. A mixed strategy p^* is an ESS for $B \iff$ the strategy profile $(p^*)^n$ is a Nash equilibrium, and no best reply $p \neq p^*$ to $(p^*)^{n-1}$ defeats or equals p^*.*

Lemma 1 clarified the structure of user populations in equilibrium. The next section applies the criterion from Corollary 2 to establish additional properties of populations in an evolutionarily stable equilibrium. In fact, these properties imply that an evolutionarily stable equilibrium is unique when it exists.

5 Uniqueness and Structure of Evolutionary Stable Strategies

We analyze the structure of evolutionary equilibria and show the uniqueness of ESS. For the first point, our focus is on the allocation of tasks to links that are consistent with evolutionary stability. Such results tell us how the structure of the network shapes evolutionary dynamics. They can be helpful for the development of algorithms calculating an ESS for a given system. The next theorem shows that in an evolutionary equilibrium there is minimal overlap in the tasks served by different links, in that two distinct links (even with the same speed) may not be used by tasks with different weights. In fact the result is stronger in that if link ℓ is used for task w and ℓ' for $w' \neq w$, then at least one of the links must not be optimal for the other link's task. This specialization result can be regarded as a stronger version of Lemma 1(6), where ℓ and ℓ' can have the same speed. Unfortunately, the specialization condition of the Theorem is necessary but not sufficient, as Observation 2 will show.

The idea of the proof of the next theorem is that if two distinct links ℓ and ℓ' are used with a probability > 0 by users with different tasks, it is possible to create a "better" mutant distribution. The mutant distribution increases the load on one of the two links, say ℓ (by putting the task with the bigger weight with a larger probability onto ℓ, and, in turn, by putting the smaller task with smaller probability onto ℓ), but uses the link overall with a smaller probability. Note that this strategy is only possible if we have different task weights.

Theorem 2 (Specialization). *Let B be a symmetric Bayesian routing game with mixed strategy p^*. Assume $w \neq w'$, $\ell \neq \ell'$ with $c_\ell \geq c_{\ell'}$, and suppose the following conditions are fulfilled.*

1. *$w \in \text{support}_\ell(p^*)$ and $w' \in \text{support}_{\ell'}(p^*)$,*
2. *$w, w' \in \text{opt}_\ell(p^*)$, and $w, w' \in \text{opt}_{\ell'}(p^*)$.*

Then there is a mutation p that defeats p^*, and hence p^* is not evolutionarily stable.

The next observation gives a counterexample showing that the specialization condition from Theorem 2 is necessary but unfortunately not sufficient for an ESS.

Observation 2. *There exists a symmetric Bayesian routing game B with a strategy p such that p meets the specialization condition of Theorem 2 for any $w \neq w'$ and $\ell \neq \ell'$, but p is not an ESS.*

Proof. Assume three resources ℓ_1, ℓ_2, ℓ_3 with speeds $c_{\ell_1} = 6$, $c_{\ell_2} = 4$, and $c_{\ell_3} = 2$. We have two users and two task sizes $w = 21$ and $w' = 1$. We define $\mu(21) = 2/3$ and $\mu(1) = 1/3$. The strategy p with $p_{\ell_1}(21) = 19/21$, $p_{\ell_2}(21) = 2/21$, $p_{\ell_2}(1) = 1/3$, and $p_{\ell_3}(1) = 2/3$ defines a symmetric Nash equilibrium fulfilling the conditions of Theorem 2. But strategy p' with $p'_{\ell_1}(21) = 19/21 - 0.001$, $p'_{\ell_2}(21) = 2/21 + 0.001$, $p'_{\ell_2}(1) = 1/3 - 0.008$, and $p'_{\ell_3}(1) = 2/3 + 0.008$ constitutes a successful mutation.

Theorem 2 is the last result required to establish the uniqueness of an ESS for symmetric Bayesian routing games.

Theorem 3 (Uniqueness). *Let B be a symmetric Bayesian routing game with ESS p^*.*

1. *Fix any two links $\ell \neq \ell'$ with the same speed, i.e. $c_\ell = c_{\ell'}$. Then for all task weights w we have $p^*_\ell(w) = p^*_{\ell'}(w)$ and $|support_\ell(p^*)| \leq 1$.*
2. *The ESS p^* is the unique ESS for B.*

Now we give a structural condition that is sufficient for an ESS. It can be used to construct an ESS in a wide variety of models where the ESS exists. Theorem 2 shows that an ESS requires links to "specialize" in tasks where distinct links do not share two distinct tasks. A stronger condition is to require that if a link is optimal for two distinct tasks, then no other link is optimal for either of the tasks. We call such a distribution clustered.

Definition 5. *In an n-player symmetric Bayesian routing game B, a symmetric strategy profile p^n is **clustered** if for any two distinct tasks w, w' and any link ℓ, if ℓ is optimal for both w and w', then no other link is optimal for w or w'.*

The next theorem establishes that every clustered symmetric Nash equilibrium is an ESS. The proof can be found in the full version.

Theorem 4 (Clustering). *Every clustered Nash equilibrium is an ESS, but not vice versa. More precisely:*

1. *Let B be a symmetric Bayesian routing game. If $(p^*)^n$ is a clustered Nash equilibrium in B, then p^* is an ESS in B.*
2. *There is a symmetric Bayesian routing game B that has a non-clustered ESS and no clustered ESS.*

Acknowledgements. We thank Funda Ergun for helpful discussions and Tom Friedetzky for the example in Observation 2. We also thank the anonymous reviewer who simplified the proof of Theorem 1.

References

1. Berenbrink, P., Goldberg, L.A., Goldberg, P., Martin, R.: Utilitarian Resource Assignment. Journal of Discrete Algorithms 4(4), 567–587 (2006)
2. Christodoulou, G., Koutsoupias, E.: The price of anarchy of finite congestion games. In: Proc. of the 37th Annual Symposium on Theory of Computing (STOC), pp. 67–73 (2005)
3. Czumaj, A., Vöcking, B.: Tight Bounds for Worst-Case Equilibria. In: Proc. 13th Annual Symposium on Discrete Algorithms (SODA), pp. 413–420 (2002)
4. Ely, J.C., Sandholm, W.H.: Evolution in Bayesian games I: Theory. Games and Economic Behavior 53(1), 83–109 (2005)
5. Etessami, K., Lochbihler, A.: The Computational Complexity of Evolutionarily Stable Strategies. Electronic Colloquium on Computational Complexity 55 (2004)
6. Fabrikant, A., Luthra, A., Maneva, E.N., Papadimitriou, C.H., Shenker, S.: On a network creation game. In: Proc. of 22nd Symposium on Principles of Distributed Computing (PODC 2003), pp. 347–351 (2003)
7. Fischer, S., Vöcking, B.: On the Evolution of Selfish Routing. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 323–334. Springer, Heidelberg (2004)
8. Gairing, M., Monien, B., Tiemann, K.: Selfish Routing with Incomplete Information. In: Proceedings of the 18th Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 1–20 (2002)
9. Harsanyi, J.C.: Games with Incomplete Information Played by 'Bayesian Players', Parts I, II, and III. In: Management Science, vol. 14, pp. 159–182, 320–334, 486–502 (1967)
10. Kearns, M.S., Suri, S.: Networks preserving evolutionary equilibria and the power of randomization. In: ACM Conference on Electronic Commerce, pp. 200–207. ACM, New York (2006)
11. Kontogiannis, S., Spirakis, P.: The Contribution of Game Theory to Complex Systems. In: Bozanis, P., Houstis, E.N. (eds.) PCI 2005. LNCS, vol. 3746, pp. 101–111. Springer, Heidelberg (2005)
12. Koutsoupias, E., Papadimitriou, C.H.: Worst-Case Equilibria. In: Meinel, C., Tison, S. (eds.) STACS 99. LNCS, vol. 1563, pp. 404–413. Springer, Heidelberg (1999)
13. Mavronicolas, M., Spirakis, P.: The Price of Selfish Routing. In: Proc. of the 33rd Annual Symposium on Theory of Computing (STOC), pp. 510–519 (2001)
14. Smith, J.M.: Evolution and the Theory of Games. Cambridge University Press, Cambridge (1982)
15. Nash, J.F.: Equilibrium Points in N-Person Games. Proc. of the National Academy of Sciences of the United States of America, 36, 48–49
16. Roughgarden, T., Tardos, É.: How Bad is Selfish Routing? Journal of the ACM 49(2), 236–259 (2002)
17. Sandholm, W.H.: Evolutionary Implementation and Congestion Pricing. Review of Economic Studies 69, 667–689 (2002)
18. van Damme, E.: Stability and Perfection of Nash Equilibria, 2nd edn. Springer, Berlin (1991)
19. Weibull, J. (ed.): Evolutionary Game Theory. The M.I.T. Press, Cambridge, MA (1995)

Convergence to Equilibria in Distributed, Selfish Reallocation Processes with Weighted Tasks

Petra Berenbrink[1,⋆], Tom Friedetzky[2,⋆⋆], Iman Hajirasouliha[1], and Zengjian Hu[1]

[1] School of Computing Science, Simon Fraser University
Burnaby, B.C., V5A 1S6, Canada
[2] Department of Computer Science, Durham University
Durham, DH1 3LE, United Kingdom

Abstract. We consider the problem of dynamically reallocating (or re-routing) m weighted tasks among a set of n uniform resources (one may think of the tasks as selfish agents). We assume an arbitrary initial placement of tasks, and we study the performance of distributed, natural reallocation algorithms. We are interested in the time it takes the system to converge to an equilibrium (or get close to an equilibrium).

Our main contributions are *(i)* a modification of the protocol in [2] that yields faster convergence to equilibrium, together with a matching lower bound, and *(ii)* a non-trivial extension to weighted tasks.

1 Introduction

We consider the problem of dynamically reallocating (or re-routing) m weighted tasks among a set of n uniform resources (one may think of the tasks as selfish agents). We assume an arbitrary initial placement of the tasks, and we study the performance of distributed, natural reallocation algorithms that does not need any global knowledge, like the number or the total weight of the tasks. We are interested in the time it takes the system to converge to an equilibrium (or get close to an equilibrium). The task in our model are to be considered as individuals, and their actions are selfish. There is no coordination whatsoever between tasks; in particular, there is no global attempt at optimising the overall situation.

We study variants of the following natural, distributed, selfish protocol: in each step, every task chooses one resource at random. It then compares the "load" of its current host resource with the "load" of the randomly chosen resource, where "load" measures not only the number of tasks on a given resource but, where appropriate, takes also into account their weights. If the load difference is

⋆ Partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC) discovery grant 250284-2002.

⋆⋆ Partially supported by U.K. Engineering and Physical Sciences Research Council (EPSRC) First Grant EP/E029124/1.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 41–52, 2007.

sufficiently large then the task will migrate to the other resource with a certain probability. Notice that migrations happen in parallel.

We express our results in terms of *Nash equilibria* or variations thereof. In general, a *Nash equilibrium* among a set of selfish users is a state in which users do not have an incentive to change their current decisions (i.e., allocation to a particular resource in our setting). We shall also consider a notion of "closeness" to Nash equilibria (namely ϵ-Nash equilibria) which describes states where users cannot improve their "cost" by a given multiplicative factor by changing their decisions.

1.1 Related Work

The papers [2,5,10,6] are most closely related to this work. In [5], Even-Dar et al. introduce the idea of using a potential function to measure closeness to a balanced allocation. They use it to show convergence for sequences of randomly-selected "best response" moves in a more general setting in which tasks may have variable weights and resources may have variable capacities. The notion of best-response moves makes it necessary for them to consider only strictly sequential protocols.

Goldberg [10] considers a protocol in which tasks select alternative resources at random and migrate if the alternative load is lower. The protocol may be implemented in a weakly distributed sense, requiring that migration events take place one at a time, and costs are updated immediately.

Even-Dar and Mansour [6] allow concurrent, independent reallocation decisions where tasks are allowed to migrate from resources with load above average to resources with load below average. They show that the system reaches a Nash equilibrium after expected $O(\log\log m + \log n)$ rounds. However, their protocol requires tasks to have a certain amount of global knowledge in order to make their decisions.

In [8], Fischer, Räcke, and Vöcking investigate convergence to Wardrop equilibria for both asymmetric and symmetric routing games. They prove various upper bounds on the speed of convergence as well as some lower bounds.

The paper most relevant to this one is [2] by Berenbrink et al. They also consider a strongly distributed system consisting of selfish users. However, they restrict themselves to uniform tasks. For their model, they show an $O(\log\log m + n^4)$ bound for the convergence time as well as a lower bound of $\Omega(\log\log m)$. They furthermore derive bounds on the convergence time to an approximate Nash equilibrium as well as an exponential lower bound for a slight modification of their protocol – which is possibly even more natural than the one considered for the $O(\log\log m + n^4)$ bound in that it results in a perfectly even distribution in expectation after only one step whereas the "quicker" protocol does not have this property.

Chien and Sinclair [4] study a version of the elementary step system (ESS) in the context of approximate Nash equilibria. They show that in some cases the *ϵ-Nash dynamics* may find an ϵ-Nash equilibrium where finding an exact Nash equilibrium seems to be hard (PLS-complete). Mirrokni and Vetta [12] study the

convergence behaviour of ESS as well as the distance from an equilibrium after a given number of iterations.

1.2 Model

In this section we shall briefly describe the models we are working in. Throughout this paper let $[n]$ denote $\{1, \ldots, n\}$. We assume discrete time, i.e., time steps $t \in \mathbb{N}$. We have m weighted tasks $b_1, \ldots, b_m$ and n uniform resources. Assume that $m \geq n$. Each task $b_i \in [m]$ is associated with a weight $w_i \geq 1$. Let $\Delta = \max\{w_i\}$ denote the maximum weight of any task. Let $M = \sum_{i=1}^{m} w_i$ be the total task weight. The assignment of tasks to resources is represented by a vector $(x_1(t), \ldots, x_n(t))$ in which $x_i(t)$ denotes the *load* of resource i at the end of step t, i.e., the sum of weights of tasks allocated to resource i. Let $\overline{x} = M/n$ be the average load. For any task $b \in [m]$, let r_b denote the current resource of task b.

Definition 1 (Nash equilibrium). *An assignment is a* Nash equilibrium *for task b if $x_{r_b} \leq x_j + w_b$ for all $j \in [n]$, i.e., if task b cannot improve its situation by migrating to any other resource.*

Definition 2 (ϵ-Nash equilibrium). *For $0 \leq \epsilon \leq 1$, we say a state is an* ϵ-Nash equilibrium *for task b if $x_{r_b} \leq x_j + (1+\epsilon)w_b$.*

Notice that this last definition is somewhat different from (and stronger than), say, Chien and Sinclair's in [4] where they say that (translated into our model) a state is an ϵ'-Nash equilibrium for $\epsilon' \in (0,1)$ if $(1-\epsilon')x_{r_b} \leq x_j + w_b$ for all $j \in [n]$. However, our definition captures theirs: for $\epsilon' \in (0,1)$ let $\epsilon = \frac{1}{1-\epsilon'} - 1 (> 0)$ and observe that $x_{r_b} \leq x_j + (1+\epsilon)w_b \leq (1+\epsilon)(x_j + w_b) = (1+(\frac{1}{1-\epsilon'}-1))(x_j+w_b) = \frac{x_j+w_b}{1-\epsilon'}$.

1.3 New Results

The main contributions of this paper are as follows. In Section 2 we consider weighted tasks. Theorem 1 shows that our protocol yields an expected time to converge to an ϵ-Nash equilibrium of $O(nm\Delta^3\epsilon^{-2})$. Notice that this would appear to be much worse than the $O(\log\log m + \text{poly}(n))$ bound in [2] when only considering uniform tasks (i.e., assuming $\Delta = 1$). We do, however, provide a lower bound of $\Omega(\epsilon^{-1}m\Delta)$ in Observation 3 for the case $\Delta \geq 2$. In Corollary 2 we also show convergence to a Nash equilibrium in expected time $O(mn\Delta^5)$ in the case of integer weights. For a reason for the (rather substantial) factor m instead of something smaller like $\log\log m$ in [2] we refer the reader to Section 2.3. The ideas used in our proofs are different from those in [2], the main reason being that equilibria are no longer unique and can, in fact, have very different potentials. It is therefore not possible (as it was in [2]) to "simply" analyze in terms of distance from equilibrium potential, namely zero, as there is no such thing. Instead, we show that tasks improve their induced cost when and if they can as long as the possible improvement is not too small.

In Section 3 we re-analyze our protocol specifically for uniform tasks to allow for a more direct comparison. In a first step, in Theorem 4 we show that for

uniform tasks our protocol reaches (the unique) Nash equilibrium in expected time $O(\log m + n \log n)$. This already is better than [2] for small values of m (roughly when $m/\log m < 2^{\Theta(n^4)}$), but still worse for $m \gg n$. The reason for this is mainly that we have smaller migration probabilities than [2] (factor ρ in the protocol below). These smaller probabilities have the effect that they speed up the end game (that is, once we are close to an equilibrium) at the expense of the early game, where the protocol in [2] is quicker. We first show a lower bound for the expected convergence time of our protocol (and uniform tasks) of $\Omega(\log m + n)$, and then mention (Remark 1) how to *combine* our protocol with that of [2] in order to obtain overall $O(\log \log m + n \log n)$. The idea here is mainly to run the protocol in [2] during the early game, and then later switch to our new protocol (which is faster for almost balanced systems), thus getting the best from both approaches.

2 The Reallocation Model with Weighted Tasks

We define our allocation process for weighted tasks and uniform resources. $X_1(0),\ldots,\ X_n(0)$ is the initial assignment. The transition from state $X(t) = (X_1(t),\ldots,X_n(t))$ to state $X(t+1)$ is given by the protocol below. Let $0 \le \epsilon \le 1$ and $\rho = \epsilon/8$. If the process converges, i.e. if $X(t) = X(t+1)$ for all $t \ge \tau$ for some $\tau \in \mathbb{N}$, then the system reached some ϵ-Nash equilibrium ("some" because ϵ-Nash equilibria are, in general, not unique). Our goal is to bound the number of steps it takes for the algorithm to converge, that is, to find the smallest τ with the property from above. We prove the following convergence result.

Theorem 1. *Let $\epsilon > 0$ and $\rho = \epsilon/8$. Let $\Delta \ge 1$ denote the maximum weight of any task. Let T be the number of rounds taken by the protocol in Figure 1 to reach an ϵ-Nash equilibrium for the first time. Then, $E[T] = O(mn\Delta^3(\rho\epsilon)^{-1})$.*

2.1 Notation and Preliminary Results

Definition 3 (Potential function). *For the analysis we use a standard potential function: $\Phi(x) = \sum_{i=1}^{n}(x_i - \overline{x})^2 = \sum_{i=1}^{n} x_i^2 - n\overline{x}$ (see also [2]).*

In the following we assume, without loss of generality, that the assignment is "normalized", meaning $x_1 \ge \cdots \ge x_n$. If it is clear from the context we will omit the time parameter t in $X(t) = (X_1(t),\ldots,X_n(t))$ and write $X = (X_1,\ldots,X_n)$ instead. We say task b has an *incentive* to move to resource i if $x_{r_b} \ge x_i + (1+\epsilon)w_b$ (notice that this is the condition used in line 4 of Algorithm 1). Let $Y^b = (Y^b(r_b,1),\ldots,Y^b(r_b,n))$ be a random variable with $\sum_{i=1}^{n} Y^b(r_b,i) = 1$. Y^b is an n-dimensional unit vector with precisely one coordinate equal to 1 and all others equal to 0. $Y^b(r_b,i) = 1$ corresponds to the event of task b moving from resource r_b to resource i (or staying at resource i if $i = r_b$). Let the corresponding probabilities $(P^b(r_b,1),\ldots,P^b(r_b,n))$ be given by

$$P^b(r_b,i) = \begin{cases} \frac{\rho(1-x_i/x_{r_b})}{n} & \text{if } r_b \ne i \text{ and } x_{r_b} > x_i + (1+\epsilon)w_b \\ 0 & \text{if } r_b \ne i \text{ and } x_{r_b} \le x_i + (1+\epsilon)w_b \\ 1 - \sum_{k\in[n]:x_i>x_k} P^b(i,k) & \text{if } r_b = i. \end{cases}$$

Algorithm 1. Greedy Reallocation Protocol for Weighted Tasks

1: **for** each task b in parallel **do**
2: let r_b be the current resource of task b
3: choose resource j uniformly at random
4: **if** $X_{r_b}(t) \geq X_j(t) + (1+\epsilon)w_b$ *//violation of Def2//* **then**
5: move task b from resource r_b to j with probability $\rho\left(1 - \frac{X_j(t)}{X_{r_b}(t)}\right)$

The first (second) case corresponds to randomly choosing resource i and finding (not finding) an incentive to migrate, and the third case corresponds to randomly choosing the current resource.

For $i \in [n]$, let $S_i(t)$ denote the set of tasks currently on resource i at step t. In the following we will omit t in S_i and write S_i if it is clear from the content. For $i, j \in [n]$ with $i \neq j$, let $I_{j,i}$ be the total weight of tasks on resource j that have an incentive to move to resource i, i.e., $I_{j,i} = \sum_{b \in S_j:\ x_j \geq x_i + (1+\epsilon)w_b} w_b$. Let $E[W_{j,i}] = \sum_{b \in S_j} w_b \cdot P^b(j,i) \leq x_j \cdot \rho(1 - x_i/x_j)/n = \rho(x_j - x_i)/n$ denote the expected total weight of tasks migrating from resource j to resource i (the last inequality is true because $I_{j,i} \leq x_j$; at most all the tasks currently on j migrate to i). Next, we show three simple observations.

Observation 2

1. $\Phi(x) = \frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{k=i+1}^{n} (x_i - x_k)^2$.
2. $E[\Phi(X(t+1))|X(t) = x] = \sum_{i=1}^{n} \left(E[X_i(t+1)|X(t) = x] - \overline{x}\right)^2$ $+ \sum_{i=1}^{n} \mathrm{var}[X_i(t+1)|X(t) = x]$.
3. $\Phi(x) \leq m^2 \Delta^2$.

Proof. Part (1) is similar to Lemma 10 in [3]. The proof of Part (2) is omitted due to space limitations. For Part (3), simply check the worst case that all the m tasks are in one resource. □

2.2 Convergence to Nash Equilibrium

In this section we bound the number of time steps for the system to reach some Nash equilibrium. We first bound the expected potential change during a fixed time step t (Lemma 3). For this we shall first prove two technical lemmas (Lemma 1 and Lemma 2): bounds for $\sum_{i=1}^{n} \left(E[X_i(t+1)|X(t) = x] - \overline{x}\right)^2$ and $\sum_{i=1}^{n} \mathrm{var}[X_i(t+1)|X(t) = x]$, respectively.

Lemma 1

1. $\sum_{i=1}^{n} \left(E[X_i(t+1)|X(t) = x] - \overline{x}\right)^2 < \Phi(x) - (2-4\rho) \sum_{i=1}^{n} \sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k)$.
2. $\sum_{i=1}^{n} \left(E[X_i(t+1)|X(t) = x] - \overline{x}\right)^2 > \Phi(x) - 2 \sum_{i=1}^{n} \sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k)$.

Proof. Since $E[W_{i,j}]$ is the expected total weight migrating from resource i to j, we have $E[X_i(t+1)|X(t)=x] = x_i + \sum_{j=1}^{i-1} E[W_{j,i}] - \sum_{k=i+1}^{n} E[W_{i,k}]$; recall that we assume $x_1 \geq \cdots \geq x_n$.

To estimate $\sum_{i=1}^{n} (E[X_i(t+1)|X(t)=x] - \overline{x})^2$, we use an indirect approach by first analysing a (deterministic) load balancing process. We then use the load balancing process to show our desired result (see [3]).

We consider the following load balancing scenario: Assume that there are n resources and every pair of resources is connected so that we have a complete network. Initially, every resource $1 \leq i \leq n$ has $z_i = x_i$ load items on it. Assume that $z_1 \geq \ldots \geq z_n$. Then every pair of resources $(i,k), i < k$ concurrently exchanges $\ell_{i,k} = E[W_{i,k}] \leq \rho(x_i - x_k)/n = \rho(z_i - z_k)/n$ load items. If $i \geq k$ we assume $\ell_{i,k} = 0$. Note that the above system is similar to one step of the diffusion load balancing algorithm on a complete graph K_n. In both cases the exact potential change is hard to calculate due to the concurrent load transfers. The idea we use now is to first "sequentialize" the load transfers, measure the potential difference after each of these sub-steps, and then to use these results to get a bound on the total potential drop for the whole step.

In the following we assume that every edge $e_s = (i,k), i,k \in [n], k > i$ is labelled with weight $\ell_{i,k} \geq 0$. Note that $\ell_{i,k} = 0$ if $x_i \leq x_k$. Let $N = n(n-1)/2$ and $E = \{e_1, e_2, \ldots e_N\}$ be the set of edges sorted in increasing order of their labels. We assume the edges are sequentially activated, starting with the edge e_1 with the smallest weight. Let $z^s = (z_1^s, \ldots, z_n^s)$ be the load vector resulting after the activation of the first s edges. Note that $z^0 = (z_1^0, \ldots, z_n^0)$ is the load vector before load balancing and $z^N = (z_1^N, \ldots, z_n^N)$ is the load vector the activation of all edges. Note that $\Phi(z^0) = \Phi(x)$ since $i \in [n]$, $z_i^0 = z_i = x_i$. Moreover, by the definition of our load balancing process and since $\ell_{i,k} = E[W_{i,k}]$ we have $z_i^N = z_i + \sum_{j=1}^{i-1} \ell_{j,i} - \sum_{k=i+1}^{n} \ell_{i,k} = x_i + \sum_{j=1}^{i-1} E[W_{i,j}] - \sum_{k=i+1}^{n} E[W_{i,k}] = E[X_i(t+1)|X(t)=x]$. Hence $\Phi(z^N) = \sum_{i=1}^{n} \left(z_i^N - \overline{x}\right)^2 = \sum_{i=1}^{n} \left(E[X_i(t+1)|X(t)=x] - \overline{x}\right)^2$. Next we bound $\Phi(z^N)$. For any $s \in [N]$, let $\Delta_s(\Phi) = \Phi(z^{s-1}) - \Phi(z^s)$ be the potential drop due to the activation of edge $e_s = (i,k)$. Note that $\Phi(z^0) - \Phi(z^N) = \sum_{s=1}^{N} \left(\Phi(z^{s-1}) - \Phi(z^s)\right) = \sum_{e_s \in E} \Delta_s(\Phi)$.

Now we bound $\Delta_s(\Phi)$. Since all edges are activated in increasing order of their weights we get $l_{i,j} \leq \ell_{i,k} = \rho(z_i - z_k)/n$ for any node j that is considered before the activation of e_s. Node i has $n-2$ additional neighbours, hence the expected load that it can send to these neighbours before the activation of edge $e_s = (i,k)$ is at most $(n-2)\ell_{i,k} < \rho(z_i - z_k) - \ell_{i,k}$. This gives us $z_i^{s-1} \geq z_i - (n-2)\ell_{i,k} > z_i - \rho(z_i - z_k) + \ell_{i,k}$. Similarly, the expected load that k receives before the activation of edge $e_s = (i,k)$ is at most $\rho(z_i - z_k) - \ell_{i,k}$. Hence, $z_k^{s-1} < z_k + \rho(z_i - z_k) - \ell_{i,k}$. Thus, $\Delta_s(\Phi) = (z_i^{s-1})^2 + (z_k^{s-1})^2 - (z_i^{s-1} - \ell_{i,k})^2 - (z_k^{s-1} + \ell_{i,k})^2 = 2\ell_{i,k}(z_i^{s-1} - z_k^{s-1} - \ell_{i,k}) > (2-4\rho)\ell_{i,k}(z_i - z_k)$. Similarly, since $z_i^{s-1} < z_i$ and $z_k^{s-1} > z_k$, we get $\Delta_s(\Phi) = 2\ell_{i,k} \cdot (z_i^{s-1} - z_k^{s-1} - \ell_{i,k}) < 2\ell_{i,k}(z_i - z_k)$. Next we bound $\Phi(z^N)$. $\Phi(z^0) - \sum_{i=1}^{n} \sum_{k=i+1}^{n} 2\ell_{i,k}(z_i - z_k) < \Phi(z^N) = \Phi(z^0) - \sum_{e_s \in E} \Delta_s(\Phi) < \Phi(z^0) - \sum_{i=1}^{n} \sum_{k=i+1}^{n} (2-4\rho)\ell_{i,k}(z_i - z_k)$. Consequently, we

get $\Phi(x) - 2\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k) < \Phi(z^N) = \sum_{i=1}^{n}\Big(E[X_i(t+1)|X_i = x] - \overline{x}\Big)^2 < \Phi(x) - (2-4\rho)\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k)$. □

Now we show an upper bound for the sum of variance. The proof is omitted due to space limitations.

Lemma 2. $\sum_{i=1}^{n} \mathrm{var}[X_i(t+1)|X(t) = x] < (2-\epsilon)\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k)$. □

We show the following lemma bounding the potential change during step t. The proof is omitted due to space limitations.

Lemma 3

1. $E[\Phi(X(t+1))|X(t) = x] < \Phi(x) - \frac{\epsilon}{2}\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k)$.
2. $E[\Phi(X(t+1))|X(t) = x] > \Phi(x) - \epsilon\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k)$. □

We first show that if $\Phi(x) \geq 4n\Delta^2$, then the system potential decreases by a multiplicative factor of at least $\rho\epsilon/4$ per round expectedly (Lemma 4). We then show that whenever x is not ϵ-Nash equilibrium, every round the system potential decreases at least by an additive factor of $\rho\epsilon/(6m\Delta)$ in expectation (Lemma 5). With these two Lemmas, we are ready to show our main result (Theorem 1). The proof is omitted due to space limitations.

Lemma 4. *If $\Phi(x) \geq 4n\Delta^2$, Δ is the maximum task weight. We have* $E[\Phi(X(t+1))|X(t) = x] < (1-\rho\epsilon/4)\Phi(x)$. □

It is easy to derive the following corollary from Lemma 4.

Corollary 1. *For $t > 0$,* $E[\Phi(X(t+1))] \leq \max\{8n\Delta^2, (1-\rho\epsilon/8)\cdot E[\Phi(X(t))]\}$.

Proof. Case analysis and Lemma 4. Details are omitted due to space limitations. □

Next we show Lemma 5, which indicates that whenever the system is not at some ϵ-Nash equilibrium, the system potential decreases by an amount of $\rho\epsilon/(6m\Delta)$ in expectation during that step.

Lemma 5. *Assume that at step t the system is not at some ϵ-Nash equilibrium. We have* $E[\Phi(X(t+1))|X(t) = x] \leq \Phi(x) - \frac{\rho\epsilon}{6m\Delta}$.

Proof. Case analysis and Lemma 3(1). Details are omitted due to space limitations.

Now we are ready to prove Theorem 1.

Proof (Proof of Theorem 1). We first show that $E[\Phi(X(\tau))] \leq 8n\Delta^2$. after $\tau = 16\log m/(\epsilon\rho)$ steps By Observation 2(3), $\Phi(X(0)) \leq m^2\Delta^2$. Repeatedly using Corollary 1 we get $E[\Phi(X(\tau))] \leq \max\{8n\Delta^2, (1-\rho\epsilon/8)^\tau \cdot \Phi(X(0))\} = 8n\Delta^2$. By Markov inequality, $\Pr[\Phi(X(\tau)) > 80n\Delta^2] \leq 0.1$.

The following proof is done by a standard martingale argument similar to [2] and [11]. Let us assume that $\Phi(X(\tau)) \leq 80n\Delta^2$. Let T be the number of additional time steps for the system to reach some ϵ-Nash equilibrium after step τ and let $t \wedge T$ be the minimum of t and T. Let $V = \rho\epsilon/(6m\Delta)$ and let $Z_t = \Phi(X(t+\tau)) + Vt$. Observe that $\{Z_t\}_{t\wedge T}$ is a supermartingale since by Lemma 5 with $X(t+\tau) = x$, $E[Z_{t+1}|Z_t = z] = E[\Phi(X(\tau+t+1)\,|\,\Phi(x) = z - Vt] + V(t+1) \leq (z - Vt - V) + V(t+1) = z$. Hence $E[Z_{t+1}] = \sum_z E[Z_{t+1}|Z_t = z] \cdot \Pr[Z_t = z] \leq \sum_z z \cdot \Pr[Z_t = z] = E[Z_t]$. We obtain $V \cdot E[T] \leq E[\Phi(X(\tau+T))] + V \cdot E[T] = E[Z_T] \leq \ldots \leq E[Z_0] \leq 80n\Delta^2$. Therefore $E[T] \leq 80n\Delta^2/V = 480mn\Delta^3(\rho\epsilon)^{-1}$, and by Markov's inequality $\Pr[T > 4800mn\Delta^3(\rho\epsilon)^{-1}] < 0.1$. Hence, after $\tau + T = 16(\rho\epsilon)^{-1}\log m + 4800mn\Delta^3(\rho\epsilon)^{-1}$ rounds, the probability that the system is not at some ϵ-Nash equilibrium is at most $0.1 + 0.1 = 0.2$.

Subdivide time into intervals of $\tau + T$ steps each. The probability that the process has *not* reached an ϵ-Nash equilibrium after s intervals is at most $(1/5)^s$. This finishes the proof. □

The following corollary bounds the convergence time to a (real, non-ϵ) Nash equilibrium in the case of all weights being integers.

Corollary 2. *Assume that every task has integer weight of at least 1, and let $\epsilon = 1/\Delta$. Let T be the number of rounds taken by the protocol in Figure 7 to reach a Nash equilibrium for the first time. Then, $E[T] = O(mn\Delta^5)$.*

Proof. When Algorithm 1 terminates, for any task b and resource $i \in [n]$, we have $x_{r_b} < x_i + (1+\epsilon)w_b < x_i + w_b + 1 \leq x_i + w_b$ since $w_b \leq \Delta$ and w_b is an integer. This implies that the system is at one of the Nash equilibria. Now, setting $\epsilon = 1/\Delta$ in Theorem 1 and using $\rho = \epsilon/8 = (8\Delta)^{-1}$, we obtain the result. □

2.3 Lower Bound for the Convergence Time

We prove the following lower bound result for the convergence time of Algorithm 1. The main reason for the slow convergence is that migration probabilities (must) depend on the quotients of the involved resource loads. Intuitively, problematic cases are those where we have two resources with (large) loads that differ only slightly. With uniform tasks we would have that *all* tasks on the higher-loaded of the two would have a (small) probability to migrate. Here, bigger tasks may be perfectly happy and there may be only very few (small) tasks on the higher-loaded resource that would attempt the migration, each also with only small probability (recall that uniform tasks implies a direct correspondence between load and number of tasks).

The authors feel that it is an interesting open problem to design a protocol that requires no or only a very small amount of (global) knowledge with regards to weight distribution, average loads, and number of tasks on each resource which circumvents this problem.

Observation 3. *Let T be the first time at which $X(t)$ is the Nash equilibrium. There is a load configuration $X(0)$ that requires $E[T] = \Omega(m\Delta/\epsilon)$.*

Proof. Consider a system with n resources, n tasks of weight 1 each, and $m-n$ tasks of weight $\Delta \geq 2$ each. Let $\ell = m/n$ where m is a multiple of n. Let $X(0) = (\,(\ell-1)\Delta+2, (\ell-1)\Delta+1, \ldots, (\ell-1)\Delta+1, (\ell-1)\Delta\,)$. The perfectly balanced state is the only Nash equilibrium. Let q be the probability for the unit-size tasks in resource 1 to move to resource n (if exactly one of the two unit-sized tasks moves, the system reaches the Nash equilibrium). By Algorithm 1, we have $q = \rho \cdot 2/(n((\ell-1)\Delta+2)) = O(\epsilon/m\Delta)$ since $\ell = m/n$ and $\rho = \epsilon/8$. Note that T is geometric distributed with probability $2q(1-q)$. Thus $E[T] = 1/(2q(1-q)) = \Omega(m\Delta/\epsilon)$. □

3 Uniform Case

In this section we show convergence for Algorithm 1 for the case that all tasks are uniform (i.e., $\Delta = 1$). [2] shows that the perfectly balanced state is the unique Nash equilibrium. We set $\epsilon = 1$ in Algorithm 1, thus $\rho = 1/(8\epsilon) = 1/8$. Note that when Algorithm 1 terminates, we have $\forall i, j \in [n]$, $x_i < x_j + (1+\epsilon)\Delta = x_j + 2$. Hence the system is in (the) Nash equilibrium.

3.1 Convergence to Nash Equilibrium

Our main result in this section is as follows.

Theorem 4. *Given any initial load configuration $X(0) = x$. Let T be the number of rounds taken by the protocol in Figure 1 to reach the unique Nash equilibrium for the first time. Then, $E[T] = O(\log m + n \log n)$. Furthermore, $T = O(\log m + n \log n)$ with a probability of at least $1 - 1/n$.*

In the following, we show (in Theorem 4) that, after $O(\log m + n \log n)$ steps, Algorithm 1 terminates with high probability. This improves the previous upper bound of $O(\log \log m + n^4)$ in [2] for small values of m. In fact, we can actually combine these two protocols to obtain a tight convergence time of $O(\log \log m + n \log n)$ w.h.p.[1]. The tightness of this result can be shown by Theorem 4.2 in [2] and Observation 7.

For simplicity we assume that m is a multiple of n, the proof can easily be extended to $n \nmid m$. We first prove Lemma 6, which is a similar result to Lemma 3(1) that bounds the expected potential drop in one round. Then we show that in each round the potential drops at least by a factor of $1/64$ if the current expected system potential is larger than $2n$ (Lemma 7(1)), and at least by a factor of $1/8n$ otherwise (Lemma 7(2)). With these two lemmas, we are ready to show Theorem 4.

We will use the same potential function $\Phi(x)$ as the one in Section 2. Recall that by Observation 2(1), for an arbitrary load configuration x, $\Phi(x) = \sum_{i=1}^{n} (x_i - \overline{x})^2 = \frac{1}{n} \sum_{i=1}^{n} \sum_{k=i+1}^{n} (x_i - x_k)^2$. For resource $i, k \in [n]$, let $E[W_{i,k}]$

[1] We use *w.h.p.* (with high probability) to denote probability $\geq 1 - 1/n^{\alpha}$ for constant $\alpha > 0$.

denote the expected number of tasks being transferred from resource i to k. Note that by Algorithm 1, if $x_i - x_k \geq 2$, $E[W_{i,k}] = x_i \cdot \rho(1 - x_k/x_i)/n = \rho(x_i - x_k)/n$, otherwise $E[W_{i,k}] = 0$. Let $S_i(x) = \{k : x_i \geq x_k + 2\}$ and $E_i(x) = \{k : x_i = x_k + 1\}$. Let $\Gamma(x) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k\in S_i(x)}(x_i - x_k)^2$. Note that the bigger $\Gamma(x)$ is, the more tasks are expected to be transferred by Algorithm 1. We first show some relations between $\Gamma(x)$ and $\Phi(x)$. The proof is omitted due to space limitations.

Observation 5. *For any load configuration x, we have*

1. *If $\Phi(x) \geq n$, then $\Gamma(x) > \Phi(x)/2$.*
2. *If $\Gamma(x) < 2$, then $\Gamma(x) = \Phi(x)^2/n$ and $\Phi(x) < \sqrt{2n}$.*
3. *If $\Phi(x) < 2$, then x is Nash equilibrium.* □

We then show the following bound for the expected potential drop in one step.

Lemma 6. $E[\Phi(X(t+1))|X(t) = x] \leq \Phi(x) - \Gamma(x)/16$.

Proof. Recall that if $x_i - x_k \geq 2$, $E[W_{i,k}] = \rho(x_i - x_k)/n$, otherwise $E[W_{i,k}] = 0$. Thus, $\Gamma(x) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k\in S_i(x)}(x_i - x_k)^2 = \rho^{-1}\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k)$. Setting $\epsilon = 1$ and $\rho = 1/8$ in Lemma 3(1), we get

$$\begin{aligned} E[\Phi(X(t+1))|X(t) = x] &= \Phi(x) - \frac{1}{2}\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k) \\ &= \Phi(x) - \frac{\rho\Gamma(x)}{2} = \Phi(x) - \frac{\Gamma(x)}{16}. \end{aligned}$$

□

The following corollaries follow from Lemma 6.

Corollary 3

1. *If $\Phi(x) \geq n$, $E[\Phi(X(t+1))|X(t) = x] < (1 - 1/32)\Phi(x)$.*
2. *If $n > \Phi(x) \geq \sqrt{2n}$, $E[\Phi(X(t+1))|X(t) = x] \leq \Phi(x) - 1/8$.*
3. *If $\sqrt{2n} > \Phi(x)$, $E[\Phi(X(t+1))|X(t) = x] \leq \Phi(x) - \Phi(x)^2/(16n)$.*

Proof. Part (1) follows directly from Lemma 6 and Observation 5(1).

To prove Part (2), if $\Phi(x) \geq \sqrt{2n}$, by Observation 5(2) $\Gamma(x) \geq 2$. Then use Lemma 6 we get $E[\Phi(X(t+1))|X(t) = x] \leq \Phi(x) - 1/8$.

For Part (3), note that $E[\Phi(X(t+1))|X(t) = x] \leq \Phi(x) - \Gamma(x)/16$ by Lemma 6. Thus it is sufficient to show that $\Gamma(x) \geq \Phi(x)^2/n$. We consider two cases for different values of $\Gamma(x)$. If $\Gamma(x) \geq 2$, $\Gamma(x) > \Phi(x)^2/n$ since $\Phi(x) < \sqrt{2n}$. If $\Gamma(x) < 2$, by Observation 5(2), $\Gamma(x) = \Phi(x)^2/n$. □

Next we prove two results that bound the expected potential drop. The proofs are omitted due to space limitations.

Lemma 7. *For any $t > 0$,*

1. $E[\Phi(X(t+1))] \leq \max\{2n, (1 - 1/64)E[\Phi(X(t))]\}$.
2. $E[\Phi(X(t+1))] \leq \left(1 - \frac{1}{8n}\right) E[\Phi(X(t))]$. □

We are now ready to prove the main result in this section.

Proof (Proof of Theorem 4). We first show that $E[\Phi(X(\tau))] \leq 2n$ after $\tau = 128 \ln m$ steps. By Observation 2(3), $\Phi(X(0)) \leq m^2\Delta^2 = m^2$. Using Lemma 7(1) iteratively for τ times, we get $E[\Phi(X(\tau))] \leq \max\{2n, (1-1/64)^{\tau}\Phi(X(0))\} \leq \max\{2n, (1-1/64)^{\tau} \cdot m^2\} = 2n$. We then show that after $T = 24n \ln n$ additional steps, the system reaches Nash equilibrium w.h.p. We apply Lemma 7(2) iteratively for T times and obtain $E[\Phi(X(\tau+T)] \leq E[\Phi(X(\tau))]\left(1-\frac{1}{8n}\right)^{T} \leq n\left(1-\frac{1}{8n}\right)^{24n\ln n} < (2n)e^{-3\ln n} < \frac{1}{n}$. By Markov's inequality, $\Pr[\Phi(X(\tau+T)) \geq 2] < 1/n$. Observation 5(3) tells us that if $\Phi(X(\tau+T)) < 2$, $X(\tau+T)$ is Nash equilibrium. Hence, after $\tau + T = 128\ln m + 24n\ln n$ steps, the probability that the system is not at the Nash equilibrium is at most $1/n$. □

Remark 1. Note that we can combine Algorithm 1 and Algorithm 1 in [2] to obtain an algorithm that converges in $O(\log\log m + n\log n)$ steps. To see this, first note that by Corollary 3.9 in [2], after $T_1 = 2\log\log m$ steps, $E[\Phi(X(T_1)] \leq 18n$. Then using a similar argument as above, we can show that after $O(\log\log m + n\log n)$, the system state is at some Nash equilibrium w.h.p.

3.2 Lower Bounds

We prove the following two lower bound results which show the tightness of Theorem 4. As discussed earlier, the "slow down" ($\log m$ as opposed to the $\log\log m$ in [2]) is the result of the introduction of the factor ρ to the migration probabilities in our protocol.

Observation 6. *Let T be the first time at which $E[X(t)] \leq c$ for constant $c > 0$. There is an initial load configuration $X(0)$ that requires $T = \Omega(\log m)$.*

Proof. Consider a system with $n = 2$ resources and m uniform tasks. Let $X(0) = (m, 0)$. We first show that $E[\Phi(X(t+1)] \geq \frac{7}{8}E[\Phi(X(t))]$. By definition, $\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k) = \frac{\rho}{n}\left(\sum_{i=1}^{n}\sum_{k=i+1}^{n} I_{i,k}(1 - x_k/x_i)(x_i - x_k)\right) \leq \frac{\Phi(x)}{8}$. Hence, setting $\epsilon = 1$ in Lemma 3(2) we obtain $E[\Phi(X(t+1))|X(t) = x] \geq \Phi(x) - \epsilon\sum_{i=1}^{n}\sum_{k=i+1}^{n} E[W_{i,k}](x_i - x_k) \geq frac7\Phi(x)8$. Now similar to Lemma 7 (1) we can show that $E[\Phi(X(t+1))] \geq 7E[\Phi(X(t))]/8$. Note that $\Phi(X(0)) = m^2/2$. To make $E[X(T)] \leq c$, we need $T = \Omega(\log m)$. □

Observation 7. *Let T be the first time at which $X(t)$ is a Nash equilibrium and T^* be the upper bound for T. There is an initial load configuration $X(0)$ that in order to make $\Pr[T \leq T^*] > 1 - 1/n$, we need $T^* = \Omega(n\log n)$.*

Proof. Consider a system with n resources and $m = n$ uniform tasks. Let $X(0)$ be the assignment given by $X(0) = (2, 1, \ldots, 1, 0)$. Denote q be the probability for the tasks in resource 1 to move to resource n (if exactly one of the two tasks in resource 1 moves, the system reaches the Nash equilibrium). By Algorithm 1 (with $\rho = 1/8$), $q = 2/(2\rho n) = 1/(8n)$. Note that T is geometric distributed with probability $2q(1-q) < 1/(4n)$. Consequently, $\Pr[T > T^*] \leq (1/(4n))^{T^*}$ (since steps $1, \ldots, T^*$ all fail). Thus, to have $\Pr[T \leq T^*] > 1 - 1/n$, we need $T^* = \Omega(n\log n)$. □

Remark 2. Note that this lower bound also holds for the protocol in [2] (with $\rho = 1$).

References

1. Ackermann, H., Röglin, H., Vöcking, B.: On the Impact of Combinatorial Structure on Congestion Games. In: Proc. 47th IEEE Annual Symposium on Foundations of Computing Science (FOCS), pp. 613–622. IEEE Computer Society Press, Los Alamitos (2006)
2. Berenbrink, P., Friedetzky, T., Goldberg, L.A., Goldberg, P., Hu, Z., Martin, R.: Distributed selfish load balancing. In: Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 354–363 (2006)
3. Berenbrink, P., Friedetzky, T., Hu, Z.: A new analytical method for parallel, diffusion-type load balancing. In: Proc. 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–10 (2006)
4. Chien, S., Sinclair, A.: Convergence to Approximate Nash Equilibria in Congestion Games. In: Proc. 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 169–178 (2007)
5. Even-Dar, E., Kesselman, A., Mansour, Y.: Convergence Time to Nash Equilibria. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 502–513. Springer, Heidelberg (2003)
6. Even-Dar, E., Mansour, Y.: Fast Convergence of Selfish Rerouting. In: Proc. 16th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 772–781 (2005)
7. Fabricant, A., Papadimitriou, C.H., Talwar, K.: The Complexity of Pure Nash Equilibria. In: Proc. 36th Annual Symposium on Theory of Computing (STOC), pp. 604–612 (2004)
8. Fischer, S., Räcke, H., Vöcking, B.: Fast Convergence to Wardrop Equilibria by Adaptive Sampling Methods. In: Proc. 38th Annual Symposium on Theory of Computing (STOC), pp. 653–662. Seattle, WA (2006)
9. Goemans, M.X., Mirrokni, V.S., Vetta, A.: Sink Equilibria and Convergence. In: Proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 152–164 (2005)
10. Goldberg, P.: Bounds for the Convergence Rate of Randomized Local Search in a Multiplayer Load-balancing Game. In: Proc. 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pp. 131–140 (2004)
11. Luby, M., Randall, D., Sinclair, A.J.: Markov Chain algorithms for planar lattice structures. SIAM Journal on Computing 31, 167–192 (2001)
12. Mirrokni, V.S., Vetta, A.: Convergence Issues in Competitive Games. In: Proc. 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX-RANDOM), pp. 183–194 (2004)

Finding Frequent Elements in Non-bursty Streams*

Rina Panigrahy[1] and Dilys Thomas[2]

[1] Microsoft Research, Mountain View, CA 94043, USA
rina@microsoft.com
[2] Department of Computer Science, Stanford University, CA 94305, USA
dilys@cs.stanford.edu

Abstract. We present an algorithm for finding frequent elements in a stream where the arrivals are not bursty. Depending on the amount of burstiness in the stream our algorithm detects elements with frequency at least t with space between $\tilde{O}(F_1/t^2)$ and $\tilde{O}(F_2/t^2)$ where F_1 and F_2 are the first and the second frequency moments of the stream respectively. The latter space complexity is achieved when the stream is completely bursty; i.e., most elements arrive in contiguous groups, and the former is attained when the arrival order is random. Our space complexity is $\tilde{O}(\alpha F_1/t^2)$ where α is a parameter that captures the burstiness of a stream and lies between 1 and F_2/F_1. A major advantage of our algorithm is that even if the relative frequencies of the different elements is fixed, the space complexity decreases with the length of the stream if the stream is not bursty.

1 Introduction

Finding frequent elements in a stream is a problem that arises in numerous database, networking and data stream applications. The objective is to use a small amount of space while scanning through the stream and detect elements whose frequency exceeds a certain threshold. This problem arises in the context of data mining while computing *Iceberg Queries* [8,2,10] where the objective is to find frequently occurring attribute values amongst database records. The problem of finding *Association Rules* [1] in a dataset also looks for frequently occurring combination of attribute values. This problem is also useful in web traffic analysis for finding frequently occurring queries in web logs [4], hot list analysis [9] and in network traffic measurement for finding heavy network flows [7,16].

Formally, given a stream of n elements the objective is to find all elements with frequency at least t. Let F_i denote the i^{th} frequency moment of the data stream. If the frequency vector of the stream with m $(=F_0)$ distinct elements is

* Supported in part by NSF Grant ITR-0331640. This work was also supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422) and the following organizations: Cisco, ESCHER, HP, IBM, Intel, Microsoft, ORNL, Qualcomm, Pirelli, Sun and Symantec.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 53–62, 2007.

denoted by $(k_1, k_2, \ldots, k_m)$, i.e., the j^{th} distinct element occurs k_j times, then $F_i = \sum_j k_j^i$.

There are several algorithms to find the frequent elements. Karp *et al.* [12] and Demaine *et al.* [6] provide a deterministic algorithm to exactly find elements with frequency greater than t in a stream of length n. The algorithm requires two passes, linear time and space $O(n/t)$. The first pass is an online algorithm generalizing a well known algorithm to find the majority element [3,15], in a stream. A variant of this algorithm is provided by Manku and Motwani [14]. All these algorithms may produce false positives whose frequencies may be slightly less than t; a second pass is used to filter these out. An algorithm for tracking frequent elements in a dynamic set where elements are being inserted and deleted is presented by Cormode and Muthukrishnan [5].

A randomized algorithm for this problem, called Count-Sketch, by Charikar *et al.* [4] (see also [11]) based on hashing requires $\tilde{O}(F_2/t^2)$ space[1] where, t gives the frequency of the frequent elements of interest and F_2 is the second frequency moment of elements in the stream (where all frequencies are capped at t). Note that F_2/t^2 can be no more than n/t if all frequencies are capped at t. In particular if the non frequent elements appear only a constant number of times this is $\tilde{O}(n/t^2)$. This means that the space complexity depends on the second frequency moment and is large if most elements have a high frequency count. If the elements arrive in random order Demaine *et al.* [6] provide an algorithm that uses space $O(F_1/t^2)$. However this algorithm strongly depends on the assumption that the arrival order is strictly random and does not work in any other case.

The Count-Sketch algorithm, on the other hand works for any arrival pattern while using space $\tilde{O}(F_2/t^2)$ but fails to exploit any randomness in the arrival order; its space complexity does not change whether the elements arrive in bursts or random order. In practice however the arrival pattern in a stream is unpredictable. Although we cannot assume that the arrival order is random, it is unlikely to be entirely bursty; i.e., it is unlikely that the same element arrives in contiguous bursts. In summary there is no single algorithm that adapts to the arrival pattern of the stream.

2 Contributions

We present an algorithm that exploits any absence of burstiness in the arrival pattern. Its space complexity decreases as the arrival pattern becomes less bursty. If all the copies of an element appear in contiguous bursts it takes space $\tilde{O}(F_2/t^2)$ and at the other extreme if elements appear in random order its space complexity gracefully improves to $\tilde{O}(F_1/t^2)$. In general the space requirement lies between $\tilde{O}(F_1/t^2)$ and $\tilde{O}(F_2/t^2)$ depending on the burstiness of the arrivals. Our results

[1] Additional space proportional to the size of the output is required to store the frequent elements once detected. We ignore this from the space complexity of all algorithms.

are especially interesting in light of a recent lower bound [13] of $\Omega(F_2/t^2)$ for this problem. However, the lower bound assumes a bursty arrival pattern.

Our algorithm works by sampling certain elements from the stream and maintaining counters for these elements. The counters are decayed over time and at every increment of a counter all the other counters are decremented by a certain value. The idea of decaying counters has been used in [14] and the decrementing of all counters on an increment has been used in [6,12]. The combination of both these steps along with the sampling gives us the improved result; dropping any of the ingredients gives a higher space complexity of $\Omega(n/t)$. The space complexity of our algorithm is $\tilde{O}(\alpha F_1/t^2)$, where α captures the burstiness of the stream, and lies between 1 and F_2/F_1.

The key advantage of our algorithm is that the space complexity decreases as we are allowed to look over a larger portion of the stream. This means in typical scenarios (where there there stream is not completely bursty) for any given space we can find the frequent elements as long as we are allowed to look at a suitably large portion of the stream. Consider for example a stream of length n where the frequent element occurs in θ fraction of the positions and all the other elements occur in $\theta/2$ fraction of the positions. In this case $\tilde{O}(F_2/t^2) \approx 2/\theta$. In comparison if the burstiness is low enough our algorithm achieves space complexity close to $\tilde{O}(F_1/t^2) \approx 1/(\theta t)$, where the count t of the frequent element is equal to $n\theta$. Since t increases with the length of the stream, this means that the space complexity decreases linearly in the count of the frequent element as the length of the stream increases. Even if the stream is not perfectly random but αF_1 is comparable to $F_{2-\epsilon}$, the space complexity grows as $\tilde{O}(\frac{F_{2-\epsilon}}{t^2}) = 1/(\theta t^{\epsilon})$. Even in this case the space complexity decreases as we use a longer stream.

The burstiness parameter α can be described as follows. Pick an element e at a random position i from the stream and find how many future occurrences of that element cumulatively occur at about the density of the frequent elements. Formally, starting from the location i find the positions of the future occurrences of the element e. Let l_{i1}, l_{i2}, ... l_{ij} be the distances of the first, second, ... jth occurrences of that element after the position i ($l_{i0} = 0$). Observe that the density of occurrence in the segments starting at location i of lengths l_{i1}, l_{i2}, ... l_{ij} are $\frac{1}{l_{i1}}$, $\frac{2}{l_{i2}}$, ... $\frac{j}{l_{ij}}$ and so forth.

Definition 1. *The ρ-burst count $B(i,\rho)$ of an element e at position i in the stream is defined as the maximum number of occurrences of e starting from position i that cumulatively occur at rate at least $\frac{\rho t}{n}$; i.e., $B(i,\rho) = 1 + max\{k|\forall(j \leq k) : \frac{j}{l_{ij}} \geq \frac{\rho t}{n}\}$.*

The additive value of 1 accounts for the occurrence of element e at position i itself.

Definition 2. $F_2(\rho) = \sum_{i=1}^{n} B(i,\rho)$

Let us look at the contribution to $F_2(\rho)$ from a single element with frequency k. Look at the sum of $B(i,\rho)$ where i spans over the k occurrences. For each

occurrence, $B(i,\rho)$ is at least 1 and at most k. If all the occurrences appear contiguously then the individual values of $B(i,\rho)$ for these occurrences are $1, 2 \ldots k$ giving a total contribution of $k(k+1)/2$. On the other extreme if all occurrences are far apart with a gap of at least $n/(\rho t)$, then each $B(i,\rho)$ for this element is 1 and the total contribution is k. So $F_1 \leq F_2(\rho) \leq (F_2 + F_1)/2 \leq F_2$.

Definition 3. $\alpha(\rho) = \frac{F_2(\rho)}{F_1}$.

In the computation of $F_2(\rho)$ we consider at most t occurrences of each element. This ensures that an element that occurs more than t times contributes at most a constant to $F_2(\rho)/t^2$. Observe that $F_2(\rho) \geq F_1$; this is because $B(i,\rho)$ is at least 1. $F_2(1)$ approaches F_1 if for most positions in the stream the next occurrence of the element at that position is farther than n/t. This tends to happen as the arrival pattern tends to a random arrival pattern.

We present an algorithm that finds all elements with frequency more than t with high probability using space $\tilde{O}(F_2(\rho)/t^2)$. Our algorithm is allowed to report elements with frequency slightly less than t: it may report elements with frequency at least $(1-\rho-2\epsilon)t$. An element with frequency more than t is reported with probability at least $1-\delta$. The space requirement is $\tilde{O}(\frac{\log(1/\delta)F_2(\rho)}{\epsilon^2 t^2})$. A second pass may be used to filter out the elements with frequency less than t.

3 Algorithm

We will present an algorithm that finds one frequent element with frequency t with probability at least $1-\delta$ using space $\tilde{O}(\frac{\log(1/\delta)F_2(\rho)}{\epsilon^2 t^2})$. In this section we assume that the value of $F_2(\rho)$ is known a-priori; we will relax this assumption in the next section. For ease of exposition let us focus on one frequent element, with frequency more than t (if it helps the reader may also assume there is only one such frequent element). If there are several frequent elements all of those can be detected by appropriately adjusting the failure probability δ. Note that in the computation of $F_2(\rho)$ the frequency of all elements is capped at t.

Our algorithm essentially maintains counters for certain elements in the stream that are sampled with probability about $\Theta(1/t)$. A counter is incremented if its associated element is found in the stream. Whenever a counter is incremented a fixed value is subtracted from all other counters. A counter is also decayed at a certain rate. Whenever a counter drops below zero it is deleted.

Algorithm:

1. The algorithm maintains a set of counters where each counter is associated with a specific element. If there is a counter associated with an element e we denote that counter value by c_e.
2. While scanning the stream, sample each position at the rate of $1/(\mu t)$, where $\mu = \frac{\log(1/\delta)}{\epsilon}$. When an element e is sampled create a new counter c_e unless a counter already exists for that element. Initialize c_e to 1.

3. For every element in the stream if the element is associated with a counter increment its counter by one and decrement every other counter by d. The value of d will be specified later during our analysis.
4. Decay every counter by $\frac{\rho t}{n}$ every step.
5. Drop a counter if its value drops below zero.
6. Whenever the value of a counter exceeds $(1 - \rho - 2\epsilon)t$ output it and ignore its future occurrences in the stream.

Remark 1. *Although in our algorithm description we perform a decay and a decrement step on each counter separately this can be implemented efficiently in* $\tilde{O}(1)$ *amortized time per stream entry by keeping a global subtraction counter that is incremented by* $\frac{\rho t}{n}$ *at every step and by* d *when an element associated with an active counter is encountered. When a new counter is created, instead of initializing it by* 1 *we can initialize it by* 1 *plus the current value in the subtraction counter. The actual value of a counter is its value minus the value of the subtraction counter. When an element with an active counter is found, instead of incrementing it by* 1*, we increment it by* $1 + d$ *(since the subtraction counter has also been incremented by* d*). We maintain the counters in sorted order, and at each step we drop all the counters with value less than the value of the subtraction counter. The amortized time spent per stream element for the drops is constant as the dropping of a counter can be charged to its creation which happens at most once per stream entry.*

The following theorem bounds the space complexity of our algorithm.

Theorem 1. *The algorithm detects a frequent element, with frequency more than* t*, with probability at least* $1 - \delta$ *and uses at most* $\tilde{O}(\frac{\log(1/\delta)F_2(\rho)}{\epsilon^2 t^2})$ *counters.*

We are focusing on detecting one frequent element, say e. Let us look at all the locations in the stream where the frequent element e does not occur. Let $S = \{i_1, i_2, i_3, \ldots, i_p\}$ denote the sampled locations, of the non frequent elements (elements with frequency less than t) sampled at the rate of $1/(\mu t)$. Let q_1, q_2, $\ldots$, q_p denote the burst counts $B(i_1, \rho)$, $B(i_2, \rho)$, $\ldots$, $B(i_p, \rho)$.

Lemma 1. *The counter* c_e *of an element,* e*, sampled at position* i*, can never exceed its burst count* $B(i, \rho)$ *during its incarnation. (Note that a counter for an element can be created and deleted multiple times).*

Proof: This is evident from the definition of the burst count, $B(i, \rho)$ and the way the counters are being decayed at rate $\frac{\rho t}{n}$. Whenever the cumulative frequency of the element drops below $\frac{\rho t}{n}$ the counter will drop below zero and hence will be deleted. □

The next lemma computes a high probability upper bound on $\sum_{i=1}^{p} q_i$. Let Q denote this upper bound.

Lemma 2. *The expected value* $E[\sum_{i=1}^{p} q_i] \leq \frac{F_2(\rho)}{\mu t}$*. With probability at least* $(1 - \delta)$ *the value of* $\sum_{i=1}^{p} q_i \leq O(\frac{F_2(\rho)}{\mu t}) + t \log(\frac{1}{\delta}) = Q$.

PROOF: Each position i is sampled with probability $1/(\mu t)$ and has burst count $B(i,\rho)$. So, $E[\sum_{i=1}^{p} q_i] = E[1/(\mu t)\sum_{i=1}^{n} B(i,\rho)] = 1/(\mu t) \times F_2^{'} \leq \frac{F_2^{'}}{\mu \times t}$. The high probability part can be proved by an application of Chernoff bounds. Each q_i lies between 0 and t. By scaling them down by a factor of t it lies in $[0,1]$. A standard application of Chernoff bounds gives the result. □

In the algorithm, the decrement value d will be set to $\frac{\mu t}{Q}$ where Q is the high probability upper bound in Lemma 2.

The proof of Theorem 1 is based on the following observations: (1) The frequent element is sampled with high probability. (2) The decay and decrement in steps 3,4 of the algorithm is small so that the counter of the frequent element survives with high value.

Conditioning on sampled locations: We will condition on the sampled locations of all elements other than e, the frequent element of interest. Let S denote the set of sampled elements other than e. We will also condition on the event that $\sum_{i=1}^{p} q_i \leq Q$ which happens with high probability by Lemma 2.

Then we will show that out of the t occurrences of the frequent element, a large fraction of them are good in the sense that if any of them are to be sampled its counter would survive and it would be reported as the frequent element. Conditioned on the sampled locations of the other elements, we will use the following process to identify good locations of the frequent element of interest, say e.

We analyze the following slightly modified algorithm for updating the counter c_e, which differs only in the decrement step.

Modified decrement step for analysis:

1. As before whenever the element e is encountered its counter c_e, if present, is incremented.
2. It is decayed at rate of $\frac{\rho t}{n}$ as before in every step.
3. Whenever an element from S is found a value d is subtracted from the counter c_e if present.

Note that in the above definition the counter operations are almost the same as in the algorithm, except that the decrement operation is allowed to happen even if the counter of an element in S may have expired. Since we are subtracting a larger amount a good element by this definition will always survive in the algorithm.

All the analysis below assumes that we work with this modified algorithm.

Definition 4. Good Occurrence of a frequent element e**:** *Conditioned on a given set of sampled locations of elements other than* e*, an occurrence of* e *is considered good if on initiating the counter* c_e *at that occurrence the counter remains positive till the end of the algorithm (with the modified decrement step). Any other occurrence of* e *is called a bad occurrence of* e*.*

Given the algorithm with the modified decrement step, we can write down with each stream entry the subtraction value that it causes on the counter c_e if present;

this subtraction value is $\frac{\rho t}{n}$ (due to the decay) if the stream entry is not from the set S and $d + \frac{\rho t}{n}$ (due to the decay and the decrement) if it is from the set S. Instead of explicitly performing the decay and the decrement steps for each stream entry we may simply subtract the subtraction value associated with that stream entry from the counter c_e if it is present. The following lemma shows that the total subtraction value of all the stream entries is small.

Lemma 3. *The total subtraction value associated with all the stream entries is at most* $(\epsilon + \rho)t$.

Proof: The total subtraction value because of the decay is at most ρt. This because the stream is of size n and the decay happens at rate $\frac{\rho t}{n}$.

The total subtraction value from the decrements is no more that dQ. This is because the decrement value of d is applicable every time a an element from S is found, which happens $\sum_{i=1}^{p} q_i$ times. The total decrement value is $d \times (\sum_{i=1}^{p} q_i)$ which by Lemma 2 is at most dQ (recall that we have already conditioned on the high probability event in Lemma 2). Since $d = \frac{\epsilon t}{Q}$ it follows that the total subtraction due to the decrement step is at most ϵt. □

The following lemma shows that there are a large number of good occurrences of the frequent element.

Lemma 4. *Conditioned on S there are at least* $(1 - \rho - \epsilon)t$ *good occurrences of element e.*

Proof: Starting from the first occurrence of e, we check if that occurrence is good or bad. If it is bad, by definition this means there must be a future point in the stream where its counter is dropped below zero due to the subtraction values. We skip all the occurrences of e up to that point and proceed to the next; we will refer to the bad occurrence of e and all the skipped positions as a bad segment (in fact all the occurrences of e in the bad segment can be shown to be bad). This divides the stream into good occurrences and bad segments; any occurrence outside of a bad segment is good. Now we will show that the total number of occurrences in these bad segments is small.

Note that the number of occurrences of e in any bad segment cannot exceed the total subtraction value of the entries in that segment as otherwise the counter c_e initiated at that start of the segment would not have become negative. Since the total subtraction value is at most $(\rho + \epsilon)t$ the total number of occurrences in the bad segments is at most the same value. So the total number of good occurrences is at least $(1 - \rho - \epsilon)t$. □

The following lemma proves that the algorithm reports a frequent element with high probability.

Lemma 5. *With probability at least* $(1-\delta)$ *the frequent element e will be sampled and survive with a counter of at least* $(1 - \rho - 2\epsilon)t$.

Proof: With probability $(1 - \delta)$ the frequent element e is sampled in one of its first $\log(1/\delta) \times \mu t = \epsilon t$ good occurrences. Let s_1 denote the total subtraction

value of entries before the sample point, and s_2 denote the total subtraction value of entries after the sample point. The number of bad occurrences before the sample point is at most s_1 (because the number of occurrence in any bad segment is at most the total subtraction value in that bad segment). The total number of occurrences before the sampling point is at most $\epsilon t + s_1$. The number of occurrences after the sampling point is at least $t - (\epsilon t + s_1)$. Further after the sampling point the total subtraction value is s_2. So after the sampling and all the subtractions the counter value in the end is at least $t - (\epsilon t + s_1)$ -s_2 = $t - (\epsilon t + (s_1 + s_2)$. Since $s_1 + s_2 = (\rho + \epsilon)t$ the result follows. □

We are now ready to prove, Theorem 1.

Proof: Let us order the counters by their creation time. We will refer to the number of times a counter has been incremented as its upcount. Since every upcount causes a decrement of all the other counters by value d, the upcount of an active counter has to be at least d times the sum of the upcounts of the newer counters. Let u_i denote the upcount of the i^{th} newest counter in this order (Note that the actual counter values is going to be the upcounts minus the amounts of subtraction due to decaying and decrementing). Since each counter value is non-negative it follows that $u_i > \sum_{j=1}^{i} u_j \times d$. Define $P_i = \sum_{j=1}^{i} u_j$. Hence $P_i > (1+d)P_{i-1}$, so $P_i > (1+d)^i$. Also P_i cannot exceed the total upcount, which is bounded by Q. The total upcount, which is at most $Q = O(\frac{F_2(\rho)}{\mu t}) + t\log(\frac{1}{\delta})$, giving $(1+d)^i < Q$. Approximating $\log(1+d) = d$ for small d, we get $i < (1/d)\log Q = \frac{F_2(\rho)}{\mu\epsilon t^2}\log Q = O(\frac{\log(1/\delta)F_2(\rho)}{\epsilon^2 t^2}\log(\frac{\log(1/\delta)F_2(\rho)}{\epsilon t} + \log(1/\delta))) = \tilde{O}(F_2(\rho)/t^2)$ □

So far we have focussed on only finding one frequent element. By adjusting the high probability parameter δ we can find all elements with frequency more than t. Since there are at most n/t such elements by replacing δ by $\frac{\delta t}{n}$ all such elements can be detected with probability $1-\delta$ by using space $\tilde{O}(\frac{\log(\frac{n}{t\delta})F_2(\rho)}{\epsilon^2 t^2})$.

So far in our analysis we have assumed that all the frequencies are bounded by t. To eliminate this assumption we note that a counter value is never allowed to exceed t as once this happens the element is reported by the algorithm and all its future occurrences are ignored.

4 Algorithm Without Assuming Knowledge of $F_2(\rho)$

We will now show how to run the algorithm without assuming the value of $F_2(\rho)$ is known.

Maintain a global counter q, initialized to zero which is incremented whenever an element with an active counter is encountered in the stream; q tracks the total upcount of all the counters. Note that $q < Q$ with high probability (see Lemma 2). The decrement amount d, which earlier depended on $F_2(\rho)$ is now set to $\frac{\epsilon' \epsilon t}{q\log^{1+\epsilon'}(q)}$.

Algorithm:

1. The algorithm maintains a set of counters, where each counter is associated with a specific element. If there is a counter associated with an element e we denote that counter value by c_e.
2. While scanning the stream, sample each position at the rate of $1/(\mu t)$, where $\mu = \frac{\log(1/\delta)}{\epsilon}$. When an element e is sampled create a new counter, c_e, unless a counter already exists for that element. Initialize c_e to 1.
3. For every element in the stream if the element is associated with a counter, increment its counter by one, decrement every other counter by $d = \frac{\epsilon^{'} \epsilon t}{q \log^{1+\epsilon^{'}}(q)}$ (Here, $\epsilon^{'}$ is any small constant), and increment q.
4. Decay every counter by $\frac{\rho t}{n}$ in every step.
5. Drop a counter if its value drops below zero.
6. Whenever the value of a counter exceeds $(1 - \rho - 2\epsilon)t$ output it and ignore its future occurrences in the stream.

The proof of correctness of the improved algorithm is very similar to the previous one except for the following differences.

Lemma 6. *The total subtraction amount due to the decrements is at most* ϵt.

Proof: The subtraction when the upcount is q is $\frac{\epsilon^{'} \epsilon t}{q \log^{1+\epsilon^{'}}(q)}$. The total subtraction is therefore at most $\sum_{q=2}^{\infty} \frac{\epsilon^{'} \epsilon t}{q \log^{1+\epsilon^{'}}(q)} \leq \int_2^{\infty} \frac{\epsilon^{'} \epsilon t}{q \log^{1+\epsilon^{'}}(q)} dq \leq \int_1^{\infty} \frac{\epsilon^{'} \epsilon t}{x^{1+\epsilon^{'}}} dx \leq (\epsilon \times t)$ □

Theorem 2. *The total number of counters used by the improved algorithm is at most* $\tilde{O}(\frac{F_2(\rho)}{\mu \epsilon^{'} \epsilon t^2})$.

Proof: The decrement value d is at least $\frac{\epsilon^{'} \epsilon t}{Q \log^{1+\epsilon^{'}}(Q)}$. Just as in the proof of Theorem 1 the number of counters is at most $(1/d)\log(Q) \leq \tilde{O}(\frac{F_2(\rho)}{\mu \epsilon \epsilon^{'} t^2})$ □

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. of the Intl. Conf. on Very Large Data Bases, pp. 487–499 (1994)
2. Beyer, K., Ramakrishnan, R.: Bottom-up computation of sparse and iceberg cubes. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pp. 359–370 (1999)
3. Boyer, R., Moore, S.: MJRTY - a fast majority vote algorithm. In: U. Texas Tech report (1982)
4. Charikar, M., Chen, K., Colton, M.F.: Finding frequent items in data streams. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 693–703. Springer, Heidelberg (2002)

5. Cormode, G., Muthukrishnan, S.: What's hot and what's not: Tracking most frequent items dynamically. In: Proc. of the ACM Symp. on Principles of Database Systems (2003)
6. Demaine, E., Ortiz, A.L., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Proceedings of the 10th Annual European Symposium on Algorithms, pp. 348–360, Sept (2002)
7. Estan, Verghese, G.: New directions in traffic measurement and accounting. In: ACM SIGCOMM Internet Measurement Workshop, ACM Press, New York (2001)
8. Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.: Computing iceberg queries efficiently. In: Proc. of 24th Intl. Conf. on Very Large Data Bases, pp. 299–310 (1998)
9. Gibbons, P.B., Matias, Y.: Synopsis data structures for massive data sets. In: DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on Eternal Memory Algorithms and Visualization, vol. A. AMS, Providence, R.I., 39.70 (1999)
10. Han, J., Pei, J., Dong, G., Wang, K.: Efficient computation of iceberg cubes with complex measures. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pp. 1–12 (2001)
11. Indyk, P., Woodruff, D.: Optimal approximations of the frequency moments of data streams. In: Proceedings of the 37th Annual ACM Symposium on Theory of computing, pp. 22–24. ACM Press, New York (2005)
12. Karp, R., Shenker, S., Papadimitriou, C.: A simple algorithm for finding frequent elements in streams and bags. In: Proc. of the ACM Trans. on Database Systems, pp. 51–55 (2003)
13. Kumar, R., Panigrahy, R.: Instance specific data stream bounds. In: Manuscript (2007)
14. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proc. of the Intl. Conf. on Very Large Data Bases (2002)
15. Misra, J., Gries, D.: Finding repeated elements. In: Science of Computer Programming (1982)
16. Pan, R., Breslau, L., Prabhakar, B., Shenker, S.: Approximate fairness through differential dropping. In: ACM SIGCOMM Computer Communication Review, pp. 23–39. ACM Press, New York (2003)

Tradeoffs and Average-Case Equilibria in Selfish Routing

Martin Hoefer[1,⋆] and Alexander Souza[2]

[1] Department of Computer and Information Science,
University of Konstanz, Germany
hoefer@inf.uni-konstanz.de
[2] Department of Computer Science, University of Freiburg, Germany
souza@informatik.uni-freiburg.de

Abstract. We consider the price of selfish routing in terms of tradeoffs and from an average-case perspective. Each player in a network game seeks to send a message with a certain length by choosing one of several parallel links that have transmission speeds. A player desires to minimize his own transmission time (latency). We study the quality of Nash equilibria of the game, in which no player can decrease his latency by unilaterally changing his link. We treat two important aspects of network-traffic management: the influence of the total traffic upon network performance and fluctuations in the lengths of the messages. We introduce a probabilistic model where message lengths are random variables and evaluate the *expected price of anarchy* of the game for various social cost functions.

For total latency social cost, which was only scarcely considered in previous work so far, we show that the price of anarchy is $\Theta\left(\frac{n}{t}\right)$, where n is the number of players and t the total message-length. The bound states that the relative quality of Nash equilibria in comparison with the social optimum increase with increasing traffic. This result also transfers to the situation when fluctuations are present, as the expected price of anarchy is $O\left(\frac{n}{\mathbb{E}[T]}\right)$, where $\mathbb{E}[T]$ is the expected traffic. For maximum latency the expected price of anarchy is even $1 + o(1)$ for sufficiently large traffic.

Our results also have algorithmic implications: Our analyses of the expected prices can be seen average-case analyses of a local search algorithm that computes Nash equilibria in polynomial time.

1 Introduction

Large-scale networks, e.g., the Internet, usually lack a central authority to coordinate the network-traffic. Instead, users that seek to send messages behave selfishly in order to maximize their own welfare. This selfish behaviour of network users motivates the use of game theory for the analysis of network-traffic.

⋆ Supported by DFG Research Training Group "Explorative Analysis and Visualization of Large Information Spaces".

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 63–74, 2007.

In standard non-cooperative games, each user, referred to as a *player*, is aware of the behaviour of other players, and seeks to minimize his own cost. A player is considered to be satisfied with his behaviour (also referred to as his strategy) if he can not decrease his cost by unilaterally changing his strategy. If all players are satisfied, then the system is said to be in a *Nash equilibrium*.

In order to relate selfishly obtained solutions with those of an (imaginary) central authority, it is necessary to distinguish between the cost of the individual players and the cost of the whole system. The latter is also referred to as *social cost*. Depending on the choice of a social cost function selfish behaviour of the players might not optimize the social cost. Consequently, the question arises how bad the social cost of a Nash equilibrium can be in comparison to the optimum. In a seminal work, Koutsoupias and Papadimitriou [1] formulated the concept of the *price of anarchy* (originally referred to as *coordination ratio*) as the maximum ratio of the social cost of a Nash equilibrium and the optimum social cost, taken with respect to all Nash equilibria of the game. In other words, the worst selfish solution in comparison with the optimum.

Specifically, in [1], the KP-model for selfish routing was introduced: each of n players seeks to send a message with respective length t_j across a network consisting of m parallel links having respective transmission speed s_i. The cost of a player j, called his *latency* ℓ_j, is the total length of messages on his chosen link i scaled with the speed, i.e., $\ell_j = \frac{1}{s_i} \sum_{k \text{ on } i} t_k$. The latency corresponds to the duration of the transmission when the channel is shared by a certain set of players. The social cost of an assignment was assumed to be the maximum duration on any channel, i.e., the social cost is $\ell_{\max} = \max_j \ell_j$. Koutsoupias and Papadimitriou [1] proved initial bounds on the price of anarchy for special cases of this game, but Czumaj and Vöcking [2] were the first ones to give tight bounds for the general case: $\Theta\left(\frac{\log m}{\log \log \log m}\right)$ for mixed and $\Theta\left(\frac{\log m}{\log \log m}\right)$ for pure Nash equilibria.

In this paper, we mainly concentrate on total latency social cost $\sum_j \ell_j$, but also apply our techniques to maximum latency. The KP-model is usually associated with maximum latency. Other latency functions were considered in [3,4,5], specifically total latency was included in [4], but there was no treatment of the price of anarchy. The price of anarchy in more general atomic and non-atomic congestion games was considered e.g. by [6,7].

Furthermore, we consider two important aspects of network-management: the influence of the *total traffic* $t = \sum_j t_j$ upon the overall system performance and *fluctuations* in the length of the respective message-lengths t_j. We model the first aspect as follows: an adversary is allowed to specify the task-lengths t_j subject to the constraint that $\sum_j t_j = t$, where t is a parameter specified in advance. We consider the price of anarchy and stability of the game with a malicous adversary explicitly restricted in this way. This model is closely related to the work of Awerbuch et al. [8], which treats the KP-model with link restrictions and unrelated machines, i.e., each task j has an arbitrary length $t_{i,j}$ on machine i. They proved tight bounds on the price of anarchy in both models that depend

on the tradeoff-ratio of the longest task with the optimum value. However, their results are only for maximum latency.

For the second aspect, fluctuations, consider a sequence of such network games all with the same set of players and the same network topology, but where the message-lengths of the players might differ from game to game. This corresponds to the natural situation that users often want to transmit messages of differing lengths. We model this by assuming that message-lengths are random variables T_j. The main question addressed in this respect is how the price of anarchy behaves in a "typical" game. To this end, we evaluate the quality of Nash equilibria in this probabilistic KP-model with the *expected value of the price of anarchy* of the game. The notion of an expected price of anarchy was, to our knowledge, considered before only by Mavronicolas et al. [9] in the context of cost mechanisms, who referred to it as *diffuse price of anarchy*. Different forms of randomization in the KP-model were either subject to machine assignment in mixed Nash equilibria [1] or subject to incomplete information [10].

We discuss our contribution in further detail below. Summarizing, we provide a connection between algorithmic game theory and average-case analysis.

1.1 Model and Notation

We consider the following network model: m parallel links with speeds $s_1 \geq \cdots \geq s_m \geq 1$ connect a source s with a target t. We note that the assumption of $s_m \geq 1$ is without loss of generality, as speeds can be normalized without changing the results. There are n players in the game, and each player seeks to send a message from s to t, where each message j has *length* t_j.

This model can naturally be described with scheduling terminology, and we refer to it as *selfish scheduling*. Each of the m links is a machine with speed s_i and each message j is a task with task-length t_j. The strategy of a player is to choose of one of the machines to execute its task. The total length on machine i is its *load* $w_i = \sum_{k \text{ on } i} t_k$. We assume that each machine executes its tasks in parallel, i.e., the resource is shared among the players that have chosen it. Hence, the duration of a task j is proportional to the total length on the chosen link i and its speed s_i, i.e., its *latency* is $\ell_j = \frac{1}{s_i} \sum_{k \text{ on } i} t_k = \frac{w_i}{s_i}$.

A *schedule* is any function $s : J \longrightarrow M$ that maps any element of the set $J = \{1, 2, \ldots, n\}$ of tasks to an element in the set of machines $M = \{1, 2, \ldots, m\}$. Each machine i executes the tasks assigned to it in parallel, which yields that each task j is finished at time ℓ_j. The *finishing time* of a machine i is hence given by $f_i = \frac{1}{s_i} \sum_{k \text{ on } i} t_k = \frac{w_i}{s_i}$. The disutility of each player is the latency of its task, i.e., the selfish incentive of every player is to minimize the individual latency.

A schedule is said to be in a *(pure) Nash equilibrium* if no player can decrease his latency by unilaterally changing the machine his task is processed on. More formally, the schedule s has the property that for each task j

$$f_i + \frac{t_j}{s_i} \geq f_{s(j)} \quad \text{holds for every } i. \tag{1}$$

In this paper, we restrict our attention to pure Nash equilibria, i.e., the strategy of each player is to choose one machine rather than a probability distribution over machines. It is known that any selfish scheduling game admits at least one pure Nash equilibrium, see, e.g., [11,12].

Schedules are valued with a certain *(social) cost* function cost : $\Sigma \longrightarrow \mathbb{R}_+$, where Σ denotes the set of all schedules. Notice that each Nash equilibrium is simply a schedule that satisfies the stability criterion (1). In contrast, an *optimum* schedule s^* is one which minimizes the cost function over all schedules, regardless if it is a Nash equilibrium or not. These differences in mind, it is natural to ask how much worse Nash equilibria can be compared to the optimum. The *price of anarchy* [1] relates the Nash equilibrium with highest social cost to the optimum, i.e, it denotes the fraction of the of the cost of the worst Nash equilibrium over the cost of the best possible solution. In contrast, the *price of stability* [13] relates the Nash equilibrium with lowest social cost to the optimum, i.e, it denotes the fraction of the cost of the best Nash equilibrium over the cost of the best possible solution.

One might wonder about a reasonable choice for a social cost function. Arguably, it depends on the application at hand which social cost function is advisable; see below for two alternatives.

1.2 Our Contribution

We characterize two important aspects of network-traffic management in the KP-setting: the influence of total traffic and fluctuations in message-lengths on the performance of Nash equilibria. Throughout the paper upper-case letters denote random variables and lower-case letters their outcomes, respectively constants.

Traffic and Fluctuations Model. We first concentrate on the influence of the system load upon Nash equilibrium performance. In the *traffic model*, an adversary is free to specify task-lengths $\boldsymbol{t} = (t_1, \ldots, t_n)$ subject to the constraints that $t_j \in [0, 1]$ and $\sum_j t_j = t$, where $0 < t \leq n$ is a parameter specified in advance. We evaluate the quality of Nash equilibria with prices of anarchy and stability on the induced set of possible inputs.

Then we formulate an extension to message fluctuations. In previous work, the KP-game was always deterministic, i.e. the task-lengths were fixed in advance. What happens if the task-lengths are subject to (random) fluctuations? What are prices of anarchy and stability in a "typical" instance? We capture these notions with the *fluctuations model*. Let the task-length T_j of a task j be a random variable with finite expectation $\mathbb{E}[T_j] = \mu_j$. As before, a schedule is a *Nash equilibrium* if (1) holds, i.e., if the concrete *realisations* t_j of the random variables T_j satisfy the stability criterion. Consequently, the set of schedules that are Nash equilibria is a random variable itself.

We define the *expected price of anarchy* by

$$\mathrm{EPoA}(\Sigma) = \mathbb{E}\left[\max\left\{\frac{\mathrm{cost}(S)}{\mathrm{cost}(S^*)} : \ S \in \Sigma \text{ is a Nash equilibrium}\right\}\right]$$

and the *expected price of stability* replacing max by min. The expected value is taken with respect to the random task-lengths T_j.

Social Cost Functions. We consider the social cost functions total latency $\sum_{j\in J} \ell_j$ and maximum latency $\max_{j\in J} \ell_j$.

Our main results for total latency $\sum_j \ell_j$ in Section 2 are as follows. Theorem 1 for the traffic model states that the (worst-case) prices of stability, respectively anarchy are essentially $\Theta\left(\frac{n}{t}\right)$. This means that in the worst-case, for small values for t, a Nash equilibrium judged with total latency social cost can be up to a factor $\Theta(n)$ larger than the optimum solution. However, as the total traffic t grows, the performance loss of Nash equilibria becomes less severe. For highly congested networks, i.e., for t being linear in n, Nash equilibria approximate the optimum solution within a constant factor.

It turns out that this behavior is stable also under the presence of fluctuations in message-lengths. Theorem 2 states an analogous result for the stochastic setting: the expected price of anarchy of this game is $O\left(\frac{n}{\mathbb{E}[T]}\right)$, where $T = \sum_j T_j$ is the total random traffic and $\mathbb{E}[T] = \omega\left(\sqrt{n \log n}\right)$. The result holds under relatively weak assumptions on the distributions of the T_j; even limited dependence among them is allowed. The assumption $\mathbb{E}[T] = \omega\left(\sqrt{n \log n}\right)$ is already satisfied if, for example, there is a constant lower bound on the task length of every player. Intuitively, it requires that on average there are not too many too small tasks in the game. In our opinion, it is a reasonable assumption when analyzing practical systems.

The results for maximum latency fall into a similar regime. For maximum latency $\max_j \ell_j$, it is already known (see [2]) that the price of stability is 1 and that the price of anarchy is $\Theta\left(\min\left\{\frac{\log m}{\log\log m}, \log\frac{s_1}{s_m}\right\}\right)$. However, in Theorem 4, we establish that the expected price of anarchy is $1 + \frac{m^2}{\mathbb{E}[T]}$, i.e., Nash equilibria are almost optimal solutions for sufficiently large traffic. This result is related to Awerbuch et al. [8] since their bounds also depend on a tradeoff between the largest task and the optimum solution. However, our results translate to selfish scheduling with coordination mechanisms; see Corollary 2.

Algorithmic Perspective. Our analyses of the expected prices of anarchy of the social cost functions provide average-case analyses of the algorithm by Feldmann et al. [14]. This algorithm calculates for any given initial assignment a pure Nash equilibrium – and for maximum latency this equilibrium is of smaller or equal social cost. Hence, for maximum latency the price of stability is 1. Using a PTAS for makespan scheduling [15], a Nash equilibrium can be found in polynomial time, which is a $(1+\varepsilon)$-approximation to the optimum schedule. This is an interesting fact as iterative myopic best-response of players can take $\Omega(2^{\sqrt{n}})$ steps to converge to a pure Nash equilibrium.

Note that this algorithm can certainly be used for solving classical scheduling problems. Our analysis hence gives bounds on the (expected) performance of that algorithm for these sort of problems (see Corollary 1). Remarkably, the analysis holds for machines with (possibly) different speeds, which is a novelty

over previous average-case analyses, e.g., [16,17], where only identical machines were considered.

Coordination Mechanisms. An interesting adjustment, which is also analyzed in terms of traffic and fluctuations here, concerns coordination mechanisms [18,19]: instead of processing all tasks in parallel each machine has a deterministic scheduling rule to sequence the tasks assigned to it. A player tries to minimize the completion time of his task by switching to the machine where it is processed earliest. Bounds on the price of anarchy for a variety of sequencing rules with makespan social cost were shown in [19]. We show that the bounds of Theorem 4 remain valid for this coordination mechanisms setting also.

2 Total Latency Cost

In this section, we consider the social cost function *total latency* $\text{cost}(s) = \sum_j \ell_j$. Let $p_i = p_i(s)$ be the number of players assigned to machine i and let $f_i = f_i(S) = \frac{1}{s_i} \sum_{k \text{ on } i \text{ in } s} t_k$ denote the finishing time of machine i in the schedule s. Observe that we can rewrite the social cost to $\text{cost}(s) = \sum_j \ell_j = \sum_i p_i f_i$.

2.1 Traffic Model

In this model, an adversary is free to specify task-lengths $\boldsymbol{t} = (t_1, \ldots, t_n)$ subject to the constraints that $t_j \in [0,1]$ and $\sum_j t_j = t$, where $0 < t \leq n$ is a parameter specified in advance.

Theorem 1. *Consider the selfish scheduling game on m machines with speeds $s_1 \geq \cdots \geq s_m \geq 1$, total task length $t = \sum_j t_j > 0$, where $t_j \in [0,1]$, and $\text{cost}(s) = \sum_j \ell_j$. Then we have the bounds*

$$\frac{n}{2t} \leq \text{PoS}(\Sigma) \leq \text{PoA}(\Sigma) \leq \frac{n}{t} + \frac{m^2+m}{t^2}, \quad \text{for } t \geq 2 \text{ and} \tag{2}$$

$$\text{PoA}(\Sigma) \leq n \quad \text{for general } t \geq 0. \tag{3}$$

Lemma 1. *Let s^* be an optimum schedule for the instance $\boldsymbol{t} = (t_1, t_2, \ldots, t_n)$ with speeds $s_1 \geq \cdots \geq s_m \geq 1$. Let $t = \sum_j t_j$, where $t_j \in [0,1]$. Then we have that $\text{cost}(s^*) \geq \frac{t^2}{\sum_k s_k}$.*

Proof. First observe that $t_j \leq 1$ implies $\text{cost}(s^*) = \sum_i p_i^* f_i^* \geq \sum_i (f_i^*)^2$. It is standard to derive $\text{cost}(s^*) \geq \sum_i (f_i^*)^2 \geq \sum_i \left(\frac{s_i}{\sum_k s_k} t\right)^2 = \frac{t^2}{\sum_k s_k}$. □

Lemma 2. *For every Nash equilibrium s for the selfish scheduling game on m machines with speeds $s_1 \geq \cdots \geq s_m \geq 1$, $t_j \in [0,1]$, and $t = \sum_j t_j$ we have that $\text{cost}(s) \leq \frac{n(t+m^2+m)}{\sum_i s_i}$.*

Proof. Recall that the cost of any schedule s can be written as $\text{cost}(s) = \sum_j \ell_j = \sum_i p_i f_i$, where ℓ_j denotes the latency of task j, p_i the number of players on machine i, and f_i its finishing time.

It turns out that it is useful to distinguish between fast and slow machines. A machine is *fast* if $s_i \geq \frac{1}{m}\sum_k s_k$; otherwise *slow*. Notice that this definition implies that there is always at least one fast machine.

Recall that the *load* of a machine i is defined by $w_i = \sum_{j \text{ on } i} t_j$. We rewrite the load w_i in terms of deviation from the respective ideally balanced load. Let $\overline{w}_i = \frac{s_i}{\sum_k s_k}\sum_j t_j = \frac{s_i}{\sum_k s_k}t$ and define the variables y_i by $w_i = \overline{w}_i + y_i$. Observe that $f_i = \frac{1}{s_i}(\overline{w}_i + y_i) = \frac{t}{\sum_k s_k} + \frac{y_i}{s_i}$. We use the notation $x_i = \frac{y_i}{s_i}$ as a shorthand.

Let s be any schedule for which the conditions (1) of a Nash equilibrium hold. Let us assume for the moment that we can prove the upper bound $|x_i| \leq \frac{m}{s_i}$ for those schedules. Further notice that the Nash conditions (1) imply that $f_i \leq f_1 + \frac{t_j}{s_1} \leq f_1 + \frac{1}{s_1}$, where t_j is the task length of any task j on machine i. Notice that machine 1 is fast, i.e., we have $\frac{1}{s_1} \leq \frac{m}{\sum_k s_k}$. Now we calculate and find $\text{cost}(s) = \sum_i p_i f_i \leq n\left(f_1 + \frac{1}{s_1}\right) \leq \frac{n(t+m^2+m)}{\sum_k s_k}$. It only remains to prove that the upper bound $|x_i| \leq \frac{m}{s_i}$ holds.

A machine i is *overloaded* if $y_i > 0$ (and hence also $x_i > 0$), *underloaded* if $y_i < 0$ (and hence also $x_i < 0$), and *balanced* otherwise, i.e., $y_i = x_i = 0$. Notice that $\sum_i w_i = \sum_j t_j = t$ and that $\sum_i w_i = \sum_i(\overline{w}_i + y_i) = \sum_i \frac{s_i}{\sum_k s_k}t + \sum_i y_i = t + \sum_i y_i$. This implies that $\sum_i y_i = 0$. Hence if there is an overloaded machine, there must also be an underloaded machine.

If all machines are balanced, then there is nothing to prove because $|x_i| = 0$ and the claimed bound holds. So let k be an underloaded machine and let i be an overloaded machine. Suppose that k receives an arbitrary task j from machine i, then its resulting finishing time equals $f_k + \frac{t_j}{s_k}$. The Nash conditions (1) state that $f_k + \frac{t_j}{s_k} \geq f_i$. The simple but important observation is that moving one task to an underloaded machine k turns it into an overloaded one. As $\frac{t_j}{s_k} \leq \frac{1}{s_k}$ we have $|x_k| \leq \frac{1}{s_k}$ for any underloaded machine k.

Finally, to prove $|x_i| \leq \frac{m}{s_i}$ we show a bound on the number of tasks whose removal is sufficient to turn an overloaded machine into an underloaded or balanced one. Let i be an overloaded machine and let there be u underloaded machines. Migrating (at most) u tasks from i to underloaded machines suffices to turn i into an underloaded or balanced machine. Suppose that there are at least u tasks on i, because otherwise moving the tasks on it to underloaded machines yields that i executes no task at all, and is hence clearly underloaded. Move u arbitrary tasks to the u underloaded machines by assigning one task to one underloaded machine, each. Now assume that i is still overloaded. This is a contradiction to $\sum_i y_i = 0$, because there are no underloaded machines in the system any more. Therefore i must be underloaded or balanced. As x_i equals the difference of f_i and $\frac{\sum_j t_j}{\sum_k s_k}$, we have that $|x_i| \leq \frac{u}{s_i} \leq \frac{m}{s_i}$ as each task contributes at most $\frac{1}{s_i}$ to the finishing time of machine i. The proof of the upper bound therefore proves the lemma. □

Proof of Theorem 1. The upper bound $\mathrm{PoA}(\Sigma) \leq \frac{n}{t} + \frac{m^2+m}{t^2}$ stated in (2) follows from Lemma 1 and Lemma 2. For the lower bound $\mathrm{PoS}(\Sigma) \geq \frac{n}{2t}$ we give the following instance. Let there be two unit-speed machines and an even number n of tasks. Let t be an even positive integer and define the task-lengths as follows: $t_1 = t_2 = \cdots = t_t = 1$ and $t_{t+1} = \cdots = t_n = 0$. In the (unique) optimum schedule s^* the t 1-tasks are scheduled on machine 1. All the other 0-tasks are assigned to machine 2. The value of that solution is $\mathrm{cost}(s^*) = t^2$, but we still have to prove that it is the unique optimum.

Assume that x 1-tasks and y 0-tasks are assigned to machine 1. The cost of such an assignment is given by $f(x, y) = (x + y)x + (n - x - y)(t - x)$. We are interested in the extrema of f subject to the constraints that $0 \leq x \leq t$ and $0 \leq y \leq n-t$. It is standard to prove that f is maximized for $x = \frac{t}{2}$ and $y = \frac{n-t}{2}$. Considering the gradient of the function $f(\frac{t}{2}+a, \frac{n-t}{2}+b) = \frac{nt}{2}+2a^2+2ab$ in the variables a and b yields that the global minimum is attained at the boundary of the feasible region; in specific for $x = t$ and $y = 0$. The value of the minimum is $\mathrm{cost}(s^*) = f(t, 0) = t^2$.

For every Nash equilibrium of the game assigning $\frac{t}{2}$ 1-tasks to each machine is necessary: If there are $x > \frac{t}{2}$ 1-tasks e.g. on machine 1, then there is a task that can improve its latency by changing the machine. Hence $x = \frac{t}{2}$ for every Nash equilibrium. It turns out that each feasible value for y gives $f(\frac{t}{2}, y) = \frac{nt}{2}$. Thus, $\mathrm{cost}(s) = \frac{nt}{2}$ and the lower bound $\mathrm{PoS}(\Sigma) \geq \frac{n}{2t}$.

Now we prove the upper bound $\mathrm{PoA}(\Sigma) \leq n$ stated in (3). Consider an arbitrary Nash equilibrium s for an instance of the game with task-lengths $t_1, \ldots, t_n$. In the sequel we will convert s into the solution s' in which all tasks are assigned to machine 1 and we will prove that $\mathrm{cost}(s) \leq \mathrm{cost}(s')$. (The solution s' need not be a Nash equilibrium.) Notice that $\mathrm{cost}(s') = \frac{nt}{s_1}$. Further we use the trivial lower bound $\mathrm{cost}(s) \geq \frac{t}{s_1}$ which holds because each task contributes to the total cost possibly on the fastest machine 1. Both bounds imply $\mathrm{PoS}(\Sigma) \leq n$.

It remains to prove that $\mathrm{cost}(s) \leq \mathrm{cost}(s')$. We construct a series of solutions $s = \sigma_1, \sigma_2, \ldots, \sigma_m = s'$, where σ_{i+1} is obtained from σ_i by moving all tasks on machine $i + 1$ to machine 1. Recall that $w_i = \sum_{j \text{ on } i} t_j$ and notice that w_1 is monotone increasing. Therefore, the Nash conditions $\frac{w_i}{s_i} \leq \frac{w_1}{s_1} + \frac{t_j}{s_1}$ for all tasks j on machine i continue to hold until i loses all its tasks. Since $t_j \leq w_i$, this condition implies $\frac{w_1+w_i}{s_1} - \frac{w_i}{s_i} \geq 0$, which will be important for the argument below. Now we consider the change of cost when obtaining σ_i from σ_{i-1} for $i \geq 2$. For ease of notation let p_1 and p_i, w_1 and w_i denote the number of players, respectively the load of machines 1 and i of the solution σ_{i-1}. We find $\mathrm{cost}(\sigma_i) - \mathrm{cost}(\sigma_{i-1}) = p_1 \frac{w_i}{s_1} + p_i \left(\frac{w_1+w_i}{s_1} - \frac{w_i}{s_i} \right) \geq 0$ since $\frac{w_1+w_i}{s_1} - \frac{w_i}{s_i} \geq 0$ as shown above. This completes the proof of the theorem. □

2.2 Fluctuations Model

Suppose that the T_j are random variables that take values in the interval $[0, 1]$ and that the T_j have respective expectations $\mathbb{E}[T_j]$. Notice that the T_j need *not* be identically distributed; even the following limited dependence is allowed.

We say that we deal with *martingale* T_j if the sequence $S_i = T_1 + \cdots + T_i + \mathbb{E}[T_{i+1}] + \cdots + \mathbb{E}[T_n]$ satisfies $\mathbb{E}[S_i \mid T_1, \ldots, T_{i-1}] = S_{i-1}$ for $i = 1, \ldots, n$.

Theorem 2. *Let* $T = \sum_j T_j$ *and* $\mathbb{E}[T] = \omega\left(\sqrt{n \log n}\right)$ *with martingale* $T_j \in [0,1]$. *Then the expected price of anarchy of the selfish scheduling game with* m *machines and speeds* $s_1 \geq \cdots \geq s_m \geq 1$ *is bounded by:*

$$\mathrm{EPoA}(\Sigma) \leq \left(\frac{n}{\mathbb{E}[T]} + \frac{m^2}{\mathbb{E}[T]^2}\right)(1 + o(1)).$$

This bound does not only hold in expectation but also with probability $1 - o(1)$.

Proof. First notice that for every outcome $t = \sum_j t_j$ of the random variable $T = \sum_j T_j$ we have $\mathrm{PoA}(\Sigma) \leq \min\left\{n, \frac{n}{t} + \frac{m^2+m}{t^2}\right\}$, by Theorem 1.

How large is the probability that T deviates "much" from its expected value? The differences of the martingale S_i are bounded by one: $|S_i - S_{i-1}| \leq 1$. Therefore we may apply the following Azuma-Hoeffding inequality (see e.g. [20] for an introduction): $\Pr[|S_n - S_0| \geq \lambda] \leq 2e^{-\frac{\lambda^2}{2n}}$. With the choice $\lambda = \sqrt{4n \log n}$ we have $\Pr\left[|T - \mathbb{E}[T]| \geq \sqrt{4n \log n}\right] \leq \frac{2}{n^2}$. Now we remember the assumption $\mathbb{E}[T] = \omega\left(\sqrt{n \log n}\right)$, and find

$$\begin{aligned}\mathrm{EPoA}(\Sigma) &\leq \mathbb{E}\left[\min\left\{n, \frac{n}{T} + \frac{m^2+m}{T^2}\right\}\right] \\ &\leq \frac{n}{\mathbb{E}[T] - \sqrt{4n \log n}} + \frac{m^2+m}{(\mathbb{E}[T] - \sqrt{4n \log n})^2} + n\frac{2}{n^2}\end{aligned}$$

and the proof is complete. □

2.3 Algorithmic Perspective

In this short section, we point out that Theorem 1 and Theorem 2 also have algorithmic implications. In specific, by proving upper bounds on the expected price of anarchy of selfish scheduling, we obtain an (average-case) analysis for a generic algorithm for classical scheduling.

In the classical scheduling problem, we are given m related machines with speeds $s_1 \geq \cdots \geq s_m \geq 1$ and n tasks with respective task-length t_j. The objective is to minimize the objective function $\sum_j \ell_j$, regardless if it is a Nash equilibrium or not. Let $\mathrm{cost}(s)$ and $\mathrm{cost}(s^*)$ denote the objective values of a schedule obtained by an algorithm and by an (not necessarily polynomial time) optimum algorithm. The quantity $\frac{\mathrm{cost}(s)}{\mathrm{cost}(s^*)}$ (respectively $\mathbb{E}\left[\frac{\mathrm{cost}(S)}{\mathrm{cost}(S^*)}\right]$) is called the *(expected) performance ratio* of the schedule s. We consider a natural greedy scheduling algorithm called NASHIFY due to Feldmann et al. [14] (introduced for maximum latency social cost and related machines); Theorem 1 and Theorem 2 imply the following result.

Corollary 1. *Under the respective assumptions of Theorem 1 and Theorem 2, the bounds stated therein are upper bounds for the (expected) performance ratio of the algorithm* NASHIFY *with the objective to minimize total latency.*

3 Maximum Latency Cost

In this section, we consider the social cost function $\text{cost}(s) = \ell_{\max} = \max_j \ell_j$. We study the traffic and the fluctuations model in Section 3.1. In Section 3.2 we show that the results extend to the study of a KP-model with a coordination mechanism.

3.1 Traffic and Fluctuations Model

Worst-case prices of stability and anarchy are already known [2] and only summarized here; see below for a further discussion.

Theorem 3. *In the selfish scheduling game on m machines with speeds $s_1 \geq \cdots \geq s_m \geq 1$ and $\text{cost}(s) = \ell_{\max}$ we have that $\text{PoS}(\Sigma) = 1$ and $\text{PoA}(\Sigma) = \Theta\left(\min\left\{\frac{\log m}{\log\log m}, \log\frac{s_1}{s_m}\right\}\right)$.*

Algorithm Nashify considered in Section 2.3 transforms any non-equilibrium schedule into a Nash equilibrium without increasing the social cost. This proves $\text{PoS}(\Sigma) = 1$. The tight bounds on the price of anarchy for pure Nash equilibria are due to Czumaj and Vöcking [2]. For the special case of identical machines different bounds for the price of anarchy hold. An upper bound $\text{PoA}(\Sigma) \leq 2 - \frac{2}{m+1}$ follows from Finn and Horowitz [11] and Nashify. Vredeveld [12] gave a schedule which is a Nash equilibrium where $2 - \frac{2}{m+1}$ is tight.

The following results follow with similar techniques as for Theorem 2. The proofs are omitted due to space limitations.

Theorem 4. *Consider the selfish scheduling game on m machines with speeds $s_1 \geq \cdots \geq s_m \geq 1$, total task length $t = \sum_j t_j > 0$, where $t_j \in [0,1]$, and $\text{cost}(s) = \max_j \ell_j$. Then we have the bound*

$$\text{PoA}(\Sigma) \leq 1 + \frac{m(m+1)}{t}.$$

Let $T = \sum_j T_j$ and $\mathbb{E}[T] = \omega\left(\sqrt{n \log n}\right)$ with martingale $T_j \in [0,1]$. Then the expected price of anarchy is bounded by

$$\text{EPoA}(\Sigma) \leq 1 + \frac{m^2}{\mathbb{E}[T]}(1 + o(1)).$$

This bound not only holds in expectation but also with probability $1 - o(1)$. Furthermore, these are bounds on the (expected) performance of algorithm Nashify.

3.2 Coordination Mechanisms

We observe that the results of Theorem 4 translate to selfish scheduling with coordination mechanisms as considered by Immorlica et al. [19]. In this scenario, the machines do not process the tasks in parallel, but instead, every machine i has

a (possibly different) local sequencing policy. For instance, with the SHORTEST-FIRST policy, a machine considers the tasks in order of non-decreasing length. Machine i uses its policy to order the tasks that have chosen i. This yields a completion time c_j, which is different for every task processed on i. The cost for a player is now the completion time of its task on the chosen machine. Naturally, a schedule is in a Nash equilibrium if no player can reduce his completion time by switching machines. Immorlica et al. [19] showed the following worst-case bounds on the price of anarchy.

Theorem 5. *The price of anarchy of a deterministic policy for scheduling on machines with speeds* $s_1 \geq \cdots \geq s_m \geq 1$ *and social cost makespan* $\text{cost}(s) = c_{\max} = \max_j c_j$ *is* $O(\log m)$. *The price of anarchy of the* SHORTEST-FIRST *policy is* $\Theta(\log m)$.

The proof of Theorem 4 can be adjusted to deliver the following direct corollary.

Corollary 2. *Under the respective assumptions of Theorem 4, the bounds stated therein are upper bounds for the (expected) price of anarchy for selfish scheduling with coordination mechanisms, arbitrary deterministic sequencing policies, and social cost makespan* $\text{cost}(s) = c_{\max} = \max_j c_j$.

4 Open Problems

In this paper we provided an initial systematic study of tradeoffs and average-case performance of Nash equilibria and naturally a lot of open problems remain. A direct problem concerns the generalization of the obtained results to the average-case performance of Nash equilibria in more general (network) congestion games. Furthermore, the characterization of average-case performance for mixed or correlated equilibria – especially for total latency – represents an interesting direction. The total latency social cost function has not been explored in a similar way as polynomial load or maximum latency, although in our opinion it has an appealing intuitive motivation as a social cost function. This might be due to the tight linear worst-case bounds, which can be obtained quite directly. However, our approach to a more detailed description offers an interesting perspective to study total latency in more general settings.

References

1. Koutsoupias, E., Papadimitriou, C.: Worst-case equilibria. STACS'99, 404–413 (1999)
2. Czumaj, A., Vöcking, B.: Tight bounds for worst-case equilibria. SODA'02, 413–420 (2002)
3. Czumaj, A., Krysta, P., Vöcking, B.: Selfish traffic allocation for server farms. STOC'02, 287–296 (2002)
4. Gairing, M., Lücking, T., Mavronicolas, M., Monien, B., Rode, M.: Nash equilibria in discrete routing games with convex latency functions. ICALPone'04, 645–657 (2004)

5. Gairing, M., Lücking, T., Mavronicolas, M., Monien, B: The price of anarchy for polynomial social cost. In: MFCS'04, vol. 29, pp. 574–585 (2004)
6. Awerbuch, B., Azar, Y., Epstein, A.: The price of routing unsplittable flow. In: STOC'05, pp. 57–66 (2005)
7. Roughgarden, T., Tardos, É.: How bad is selfish routing? Journal of the ACM 49(2), 236–259 (2002)
8. Awerbuch, B., Azar, Y., Richter, Y., Tsur, D.: Tradeoffs in worst-case equilibria. In: Solis-Oba, R., Jansen, K. (eds.) WAOA 2003. LNCS, vol. 2909, pp. 41–52. Springer, Heidelberg (2004)
9. Mavronicolas, M., Panagopoulou, P., Spirakis, P.: A cost mechanism for fair pricing of resource usage. In: Deng, X., Ye, Y. (eds.) WINE 2005. LNCS, vol. 3828, pp. 210–224. Springer, Heidelberg (2005)
10. Gairing, M., Monien, B., Tiemann, K.: Selfish routing with incomplete information. In: SPAA'05, vol. 17, pp. 203–212 (2005)
11. Finn, G., Horowitz, E.: A linear time approximation algorithm for multiprocessor scheduling. BIT 19, 312–320 (1979)
12. Vredeveld, T.: Combinatorial approximation algorithms. Guaranteed versus experimental performance. PhD thesis, Technische Universiteit Eindhoven (2002)
13. Anshelevich, E., Dasgupta, A., Tardos, É., Wexler, T.: Near-optimal network design with selfish agents. In: STOC'03, pp. 511–520 (2003)
14. Feldmann, R., Gairing, M., Lücking, T., Monien, B., Rode, M.: Nashification and the coordination ratio for a selfish routing game. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 514–526. Springer, Heidelberg (2003)
15. Hochbaum, D., Shmoys, D.: A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. SIAM Journal on Computing 17(3), 539 (1988)
16. Souza, A., Steger, A.: The expected competitive ratio for weighted completion time scheduling. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 620–631. Springer, Heidelberg (2004)
17. Scharbrodt, M., Schickinger, T., Steger, A.: A new average case analysis for completion time scheduling. In: STOC'02, vol. 34, pp. 170–178 (2002)
18. Christodoulou, G., Koutsoupias, E., Nanavati, A.: Coordination mechanisms. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 345–357. Springer, Heidelberg (2004)
19. Immorlica, N., Li, L., Mirrokni, V., Schulz, A.: Coordination mechanisms for selfish scheduling. In: Deng, X., Ye, Y. (eds.) WINE 2005. LNCS, vol. 3828, pp. 55–69. Springer, Heidelberg (2005)
20. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)

On the Variance of Subset Sum Estimation

Mario Szegedy[1] and Mikkel Thorup[2]

[1] Department of Computer Science, Rutgers, the State University of New Jersey, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019, USA
szegedy@cs.rutgers.edu
[2] AT&T Labs—Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA
mthorup@research.att.com

Abstract. For high volume data streams and large data warehouses, sampling is used for efficient approximate answers to aggregate queries over selected subsets. We are dealing with a possibly heavy-tailed set of weighted items. We address the question:

> Which sampling scheme should we use to get the most accurate subset sum estimates?

We present a simple theorem on the variance of subset sum estimation and use it to prove optimality and near-optimality of different known sampling schemes. The performance measure suggested in this paper is the average variance over all subsets of any given size. By optimal we mean there is no set of input weights for which any sampling scheme can have a better average variance. For example, we show that appropriately weighted systematic sampling is simultaneously optimal for all subset sizes. More standard schemes such as uniform sampling and probability-proportional-to-size sampling with replacement can be arbitrarily bad.

Knowing the variance optimality of different sampling schemes can help deciding which sampling scheme to apply in a given context.

1 Introduction

Sampling is at the heart of many DBMSs, Data Warehouses, and Data Streaming Systems. It is used both internally, for query optimization, enabling selectivity estimation, and externally, for speeding up query evaluation, and for selecting a representative subset of data for visualization [1]. Extensions to SQL to support sampling are present in DB2 and SQLServer (the TABLESAMPLE keyword [2]), Oracle (the SAMPLE keyword [3]), and can be simulated for other systems using syntax such as ORDER BY RANDOM() LIMIT 1. Users can also ensure sampling is used for query optimization, for example in Oracle (using `dynamic-sampling` [4]).

In our mathematical model the entire data stream or data base is represented by a sequence of positive weights, and the goal is to support queries to arbitrary subset sums of these weights. With unit weights, we can compute subset sizes which together with the previous sums provide the subset averages. We note that

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 75–86, 2007.

there has been several previous works in the data base community focused on sampling based subset sum estimation (see, e.g., [5,6,7]). The question addressed here is which sampling scheme we should use to get the most accurate subset sum estimates. More precisely, we study the variance of sampling based subset sum estimation.

The formal set-up is as follows. We are dealing with a set of items $i \in [n]$ with positive weights w_i. Here $[n] = \{1, ..., n\}$. A subset $S \subseteq [n]$ of these are sampled, and each sampled item i is given a weight estimate $\hat{w}_i$. Unsampled items $i \notin S$ have a zero weight estimate $\hat{w}_i = 0$. We generally assume that sampling procedures include such weight estimates. We are mostly interested in unbiased estimation procedures such that

$$\mathrm{E}[\hat{w}_i] = w_i \qquad \forall i \in [n]. \tag{1}$$

Often one is really interested in estimating the total weight w_I of a subset $I \subseteq [n]$ of the items, that is, $w_I = \sum_{i \in I} w_i$. As an estimate $\hat{w}_I$, we then use the sum of the sampled items from the subset, that is, $\hat{w}_I = \sum_{i \in I} \hat{w}_I = \sum_{i \in I \cap S} \hat{w}_i$. By linearity of expectation this is also unbiased, that is, from (1) we get

$$\mathrm{E}[\hat{w}_I] = w_I \qquad \forall I \subseteq [n]. \tag{2}$$

We are particularly interested in cases where the subset I is unknown at the time the sampling decisions are made. For example, in an opinion poll, the subset corresponding to an opinion is only revealed by the persons sampled for the poll. In the context of a huge data base, sampling is used to reduce the data so that we can later support fast approximate aggregations over arbitrary selected subsets [8,1,7].

Applied to Internet traffic analysis, the items could be records summarizing the flows streaming by a router. The weight of a flow would be the number of bytes. The stream is very high volume so we can only store samples of it efficiently. A subset of interest could be flow records of a newly discovered worm attack whose signature has just been determined. The sample is used to estimate the size of the attack even though the worm was unknown at the time the samples were chosen. This particular example is discussed in [7], which also shows how the subset sum sampling can be integrated in a data base style infrastructure for a streaming context. In [7] they use the threshold sampling from [9] which is one the sampling schemes that we will analyze below.

Before continuing, we note that there is an alternative use of sampling for subset sum estimation in data bases; namely where the whole data set is organized to generate a sample from any selected subset. Generating such samples on-the-fly has been studied for different sampling schemes in [5,10,6]. For uniform sampling [6] we create a random permutation of the data. To get a sample of size k from a subset I, we find the first k items from I in the permutation. Based on the samples and their rank in the total permutation, we can estimate the weight of I.

In this paper, we generate the sample first, forget the data, and use the sample to estimate the weight of arbitrary subsets. As discussed in [7], this is how

we have to do in a high volume streaming context where items arrive faster and in larger quantities than can be saved; hence where only a sample can be stored efficiently. Sampling first is also relevant if we want to create a reduced approximate version of a large data ware house that can be downloaded on smaller device.

Generally there are two things we want to minimize: (a) the number of samples viewed as a resource, and (b) the variance as a measure for uncertainty in the estimates.

For several sampling schemes, we already understand the optimality with respect to the sum of the individual variances

$$\Sigma V = \sum_{i \in [n]} \mathrm{Var}[\hat{w}_i] \tag{3}$$

as well as the variance of the total sum

$$V\Sigma = \mathrm{Var}[\hat{w}_{[n]}] \quad \left(= \mathrm{Var}[\sum_{i \in [n]} \hat{w}_i] \right) \tag{4}$$

However, what we are really interested in is the estimation of subsets of arbitrary sizes.

Performance Measure. The purpose of our sampling is later to be able to estimate arbitrary subset sums. With no advance knowledge of the subsets of interest, a natural performance measure is the expected variance for a random subset. We consider two distributions on subsets:

$\mathcal{S}_{m:n}$**:** denoting the uniform distribution on subsets of size m.
$\mathcal{S}_p$**:** denoting the distribution on subsets where each item is included independently with probability p.

Often we are interested in smaller subsets with $p = o(1)$ or $m = o(n)$. The corresponding expected variances are denoted

$$V_{m:n} = \mathrm{E}_{I \leftarrow \mathcal{S}_{m:n}}[\mathrm{Var}[\hat{w}_I]]$$
$$W_p = \mathrm{E}_{I \leftarrow \mathcal{S}_p}[\mathrm{Var}[\hat{w}_I]]$$

Note that $V_{1:n} = \Sigma V/n$ and $V_{n:n} = W_1 = V\Sigma$.

It is important to realize that we are considering two independent types of random subsets: the subset sampled by a sampling scheme and the subset selected for estimating. For a given selected subset, we consider the variance of random sampling based estimation of the total weight of that particular subset. We then find the expectation of this variance over a randomly selected subset. The sampling may be biased towards heavier items, but the randomly selected subset is unbiased in that each item has an equal chance of being selected.

We are not aware of any previous analysis of the average variance of subset sum estimation. As we shall discuss later, there has been theoretical studies

of $V\Sigma$ and ΣV for different sampling schemes [11,12,13]. Also, there has been experimental work evaluating sampling based subset sum estimation [11,7]. However, we believe that this is the first theoretical study of the accuracy of sampling based subset sum estimation for non-trivial subsets that are neither just singletons, nor the full set. We believe that our average variance is an important parameter to understand both in theory and in practice when we wish to decide which sampling scheme to apply in a given context.

A Basic Theorem. Our basic theorem below states that our subset sum variances $V_{m:n}$ and W_p are simple combinations of the previously studied quantities ΣV and $V\Sigma$:

Theorem 1. *For any sampling scheme, we have*

$$V_{m:n} = \frac{m}{n}\left(\frac{n-m}{n-1}\,\Sigma V + \frac{m-1}{n-1}\,V\Sigma\right) \tag{5}$$

$$W_p = p\left((1-p)\Sigma V + pV\Sigma\right). \tag{6}$$

Theorem 1 holds for arbitrarily correlated random estimators $\hat{w}_i, i \in [n]$ with $E[\hat{w}_i] = w_i$. That is, we have an arbitrary probability space Φ over functions $\hat{w}$ mapping indices $i \in [n]$ into estimates $\hat{w}_i$. Expectations and variances are all measured with respect to Φ. The only condition for our theorem to hold true is that the estimate of a subset is obtained by summing the estimates of its element, that is, $\hat{w}_I = \sum_{i\in I} \hat{w}_i$.

One nice consequence of (5) is that

$$V_{m:n} \geq m\frac{n-m}{n-1}\,V_{1:n} \tag{7}$$

This means that no matter how much negative covariance we have, on the average, it reduces the variance by at most a factor $\frac{n-1}{n-m}$.

A nice application of (6) is in connection with a random partition into q subsets where each item independently is assigned a random subset. A given subset includes each item with probability $p = 1/q$, so by linearity of expectation, the expected total variance over all sets in the partition is $q \cdot W_p = ((1-p)\Sigma V + pV\Sigma)$.

Known Sampling Schemes. We will apply Theorem 1 to study the optimality of some known sampling schemes with respect to the average variance of subset sum estimation. Below we first list the schemes and discuss. what is known about ΣV and $V\Sigma$. Our findings with Theorem 1 will be summarized in the next subsection.

Most of the known sampling schemes use Horvitz-Thompson estimators: if item i was sampled with probability p_i, it is assigned an estimate of $\hat{w}_i = w_i/p_i$. Horvitz-Thompson estimators are trivially unbiased.

For now we assume that the weight w_i is known before the sampling decision is made. This is typically not the case in survey sampling. We shall return to this point in Section 4.

Uniform Sampling Without Replacement (U−R). In uniform sampling without replacement, we pick a sample of k items uniformly at random. If item i is sampled it gets weight estimate $\hat{w}_i = w_i n/k$. We denote this scheme U-R$_k$.

Probability Proportional to Size Sampling with Replacement (P+R). In probability proportional to size sampling with replacement, each sample $S_j \in [n]$, $j \in [k]$, is independent, and equal to i with probability $w_i/w_{[n]}$. We say that i is sampled if $i = S_j$ for some $j \in [k]$. This happens with probability $p_i = 1-(1-w_i/w_{[n]})^k$. If i is now sampled, we use the Horvitz-Thompson estimator $\hat{w}_i = w_i/p_i$. We denote this scheme P+R$_k$.

We do not consider probability proportional to size sampling without replacement (P−R) because we do not know of any standard estimates derived from such samples.

Threshold Sampling (THR). The threshold sampling is a kind of Poisson sampling. In Poisson sampling, each item i is picked independently for S with some probability p_i. For unbiased estimation, we use the Horvitz-Thompson estimate $\hat{w}_i = w_i/p_i$ when i is picked.

In threshold sampling we pick a fixed threshold τ. For the sample S, we include all items with weight bigger than τ. Moreover, we include all smaller items with probability w_i/τ. Sampled items $i \in S$ have the Horvitz-Thompson estimate $\hat{w}_i = w_i/p_i = w_i/\min\{1, w_i/\tau\} = \max\{w_i, \tau\}$. With $k = \sum_i \min\{1, w_i/\tau\}$ the expected number of samples, we denote this scheme THR$_k$. Threshold sampling is known to minimize ΣV relative to the expected number of samples.

In survey sampling, one often makes the simplifying assumption that if we want k samples, no single weight has more than a fraction $1/k$ of the total weight [12, p. 89]. In that case threshold sampling is simply Poisson sampling with probability proportional to size as described in [12, p. 85–87]. More precisely, the threshold becomes $\tau = w_{[n]}/k$, and each item is sampled with probability w_i/τ. We are, however, interested in the common case of heavy tailed distributions where one or a few weights dominate the total [14,15]. The name "threshold sampling" for the general case parametrized by a threshold τ is taken from [9].

Systematic Threshold Sampling (SYS). We consider the general version of systematic sampling where each item i has an individual sampling probability p_i, and if picked, a weight estimate w_i/p_i. In contrast with Poisson sampling, the sampling decisions are not independent. Instead we pick a single uniformly random number $x \in [0, 1]$, and include i in S if and only if for some integer j, we have

$$\sum_{h<i} p_i \le j + x < \sum_{h \le i} p_i$$

It is not hard to see that $\Pr[i \in S] = p_i$. Let $k = \sum_{i \in [n]} p_i$ be the expected number of samples. Then the actual number of samples is either $\lfloor k \rfloor$ or $\lceil k \rceil$. In particular, this number is fixed if k is an integer. Below we assume that k is integer.

In systematic threshold sampling we perform systematic sampling with exactly the same sampling probabilities as in threshold sampling, and denote this scheme SYS_k. Hence for each item i, we have identical marginal distributions $\hat{w}_i$ with THR_k and SYS_k.

Priority Sampling (PRI). In priority sampling from [16] we sample a specified number of $k < n$ samples. For each item, a we generate a uniformly random number $r_i \in (0,1)$, and assign it a priority $q_i = w_i/r_i$. We assume these priorities are all distinct. The k highest priority items are sampled. We call the $(k+1)$th highest priority the threshold τ. Then i is sampled if and only if $q_i > \tau$, and then the weight estimate is $\hat{w}_i = \max\{\tau, w_i\}$. This scheme is denoted PRI_k.

Note that the weight estimate $\hat{w}_i = \max\{\tau, w_i\}$ depends on the random variable τ which is defined in terms of all the priorities. This is not a Horvitz-Thompson estimator. In [16] it is proved that this estimator is unbiased. Moreover it is proved that priority sampling with at least two samples has no covariance between different item estimates.

Variance Optimality of Known Sampling Schemes. Below we compare $V_{m:n}$ and W_p for the different sampling schemes. Unless otherwise stated, we consider an arbitrary fixed weight sequence $w_1, ...w_n$. Using Theorem 1 most results are derived quite easily from existing knowledge on ΣV and $V\Sigma$. The derivation including the relevant existing knowledge will be presented in Section 3.

When comparing different sampling schemes, we use a superscript to specify which sampling scheme is used. For example $V^{\Phi}_{m:n} < V^{\Psi}_{m:n}$ means that the sampling scheme Φ obtains a smaller value of $V_{m:n}$ than does Ψ on the weight sequence considered.

For a given set of input weights $w_1, ...w_n$, we think abstractly of a sampling scheme as a probability distribution Φ over functions $\hat{w}$ mapping items i into estimates $\hat{w}_i$. We require unbiasedness in the sense that $\mathrm{E}_{\hat{w}\leftarrow\Phi}[\hat{w}_i] = w_i$. For a given $\hat{w} \in \Phi$, the number of samples is the number of non-zeroes. For any measure over sampling schemes, we use a superscript OPT_k to indicate the optimal value over all sampling schemes using an expected number of at most k samples. For example, $V^{\mathrm{OPT}_k}_{m:n}$ is the minimal value of $V^{\Phi}_{m:n}$ for sampling schemes Φ using an expected number of at most k samples.

Optimality of SYS, THR, and PRI. For any subset size m and sample size k, we get

$$V^{\mathrm{OPT}_k}_{m:n} = V^{\mathrm{SYS}_k}_{m:n} = \frac{n-m}{n-1} V^{\mathrm{THR}_k}_{m:n} \tag{8}$$

The input weights $w_1, ..., , w_n$ were arbitrary, so we conclude that systematic threshold sampling optimizes $V_{m:n}$ for any possible input, subset size m, and sample size, against any possible sampling scheme. In contrast, threshold sampling is always off by exactly a factor $\frac{n-1}{n-m}$.

Similarly, for any subset inclusion probability p, we get that

$$W_p{}^{\mathrm{OPT}_k} = W_p{}^{\mathrm{SYS}_k} = (1-p)\, W_p{}^{\mathrm{THR}_k} \tag{9}$$

From [13], we get that

$$V_{m:n}^{\mathrm{PRI}_{k+1}} \leq V_{m:n}^{\mathrm{THR}_k} \leq V_{m:n}^{\mathrm{PRI}_k} \tag{10}$$

$$W_p^{\mathrm{PRI}_{k+1}} \leq W_p^{\mathrm{THR}_k} \leq W_p^{\mathrm{PRI}_k} \tag{11}$$

Hence, modulo an extra sample, priority sampling is as good as threshold sampling, and hence at most a factor $\frac{n-1}{n-m}$ or $1/(1-p)$ worse than the optimal systematic threshold sampling.

Anti-optimality of U−R and P+R. We argue that standard sampling schemes such as uniform sampling and probability proportional to size sampling with replacements may be arbitrarily bad compared with the above sampling schemes. The main problem is in connection with heavy tailed weight distributions where we likely have one or a few dominant weights containing most of the total weight. With uniform sampling, we are likely to miss the dominant weights, and with probability proportional to size sampling with replacement, our sample gets dominated by copies of the dominant weights. Dominant weights are expected in the common case of heavy tailed weight distributions [14,15].

We will analyze a concrete example showing that these classic schemes can be arbitrarily bad compared with the above near-optimal schemes. The input has a large weight $w_n = \ell$ and $n-1$ unit weights $w_i = 1$, $i \in [n-1]$. We are aiming at k samples. We assume that $\ell \gg n \gg k \gg 1$ and $\ell \geq k^2$. Here $x \gg y \Longleftrightarrow x = \omega(y)$. For this concrete example, in a later journal version, we will show that

$$V_{m:n}^{\mathrm{OPT}_k} \approx (n-m)m/k, \quad V_m^{\text{U-R}_k} \gtrsim \ell^2 m/k, \quad \text{and} \quad V_{m:n}^{\text{P+R}_k} \gtrsim \ell m/k$$

Here $x \approx y \Longleftrightarrow x = (1 \pm o(1))y$ and $x \gtrsim y \Longleftrightarrow x \geq (1-o(1))y$. A corresponding set of relations can be found in terms of p, replacing $n-m$ with $n(1-p)$ and m with pn. We conclude that uniform sampling with replacement is a factor ℓ^2/n from optimality while probability proportional to size sampling with replacement is a factor ℓ/n from optimality. Since $\ell \gg n$ it follows that both schemes can be arbitrarily far from optimal.

Relating to Previous Internet Experiments. From [11] we have experiments based on a stream segment of 85,680 flow records exported from a real Internet gateway router. These items were heavy tailed with a single record representing 80% of the total weight. Subsets considered were entries of an 8×8 traffic matrix, as well as a partition of traffic into traffic classes such as ftp and dns traffic. Figure 1 shows the results for the 8×8 traffic matrix with all the above mentioned sampling schemes (systematic threshold sampling was not included in [11], but is added here for completeness). The figure shows the relative error measured as the sum of errors over all 64 entries divided by the total traffic. The error is a function of the number k of samples, except with THR, where k represents the expected number of samples.

We note that U−R is worst. It has an error close to 100% because it failed to sample the large dominant item. The P+R is much better than U−R, yet much worse than the near-optimal schemes PRI, THR, and SYS. To qualify the difference, note that P+R use about 50 times more samples to get safely below a 1% relative error.

Among the near-optimal schemes, there is no clear winner. From our theory, we would not expect much of a difference. We would expect THR to be ahead of PRI by at most one sample. Also, we are studying a partitioning into 64 sets, so by Theorem 1, the average variance advantage of SYS is a factor $1 - 1/64$, which is hardly measurable.

The experiments in Figure 1 thus fit nicely with our theory even though the subsets are not random. The strength of our mathematical results is that no one can ever turn up with a different input or a new sampling scheme and perform better on the average variance.

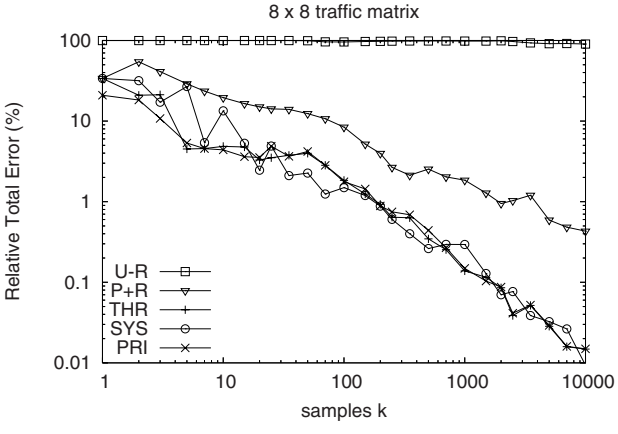


Fig. 1. Estimation of 8×8 traffic matrix

Choosing a Sampling Scheme in a Streaming Context. One of our conclusions above is that systematic threshold sampling is optimal for the average subset variances no matter the subset size m or inclusion probability p. However, in a streaming context, we have several problems with both threshold sampling and systematic threshold sampling. For both of these schemes we have some problems because we do not know the threshold in advance. Moreover, threshold sampling only gives an expected number of samples which is not so useful with a hard resource limit on the number of samples. A major problem with systematic sampling in general is that items should be shuffled in order to avoid strong correlations in the sampling of items [12, p. 92]. This is not possible in a streaming context. Priority sampling, however, is ideally suited for streaming. We just need to generate priorities for the items as they arrive, and use a standard priority queue to maintain the $k+1$ items of highest priority. Thus we are

left with priority sampling among the known near-optimal schemes. From (9) and (11), we get that

$$V_{m:n}^{\mathrm{PRI}_{k+1}} \leq \frac{n-1}{n-m} V_{m:n}^{\mathrm{OPT}_k} \tag{12}$$

Even if we don't know what the optimal appropriate scheme is, this inequality provides a limit to the improvement with any possible scheme. In particular, if k is not too small, and m is not too close to n, there is only limited scope for improvement. Priority sampling is therefore a good choice in a streaming context.

Contribution: Theory for Sake of Practice. There are no deep technical proofs in this paper. Theorem 1 and its proof in Section 2 could easily be included in a text book. The significance is the formulation of the theorem and the optimality conclusions it allows us to draw about known sampling schemes.

Having seen the formulation of Theorem 1, it is hard to argue that it wasn't obvious in the first place. When we started this research, our objective was to show that negative covariance can only provide a limited advantage for smaller sets, that is, $V_{m:n} \gtrsim m V_{1:n}$ for $m = o(n)$. We did not expect the simple inequality (7) which is tight for any set of weights. Realizing that $V_{m:n}$ could be expressed exactly in terms of the previously studied quantities ΣV and $V\Sigma$ came as a very pleasant surprise, reinforced our feeling that $V_{m:n}$ is a natural measure for comparing the performance of different sampling schemes, not only from a practical viewpoint, but also from a mathematical one.

The significance of our simple theorem is the importance of the conclusions it allows us to draw about concrete sampling schemes. As we saw in Section 1, this can help us choosing a sampling scheme in a specific practical context. Here it is important that Theorem 1 provides an exact answer rather than just asymptotic bounds. This paper is thus not just a presentation of a simple theorem. It also makes an effort to demonstrate potential impact on real world practice.

Contents. The rest of the paper is divided as follows: In Section 2 we prove Theorem 1. In Section 3 we will derive the optimality results. Finally we discuss some extensions in Section 4.

2 Proof of the Basic Theorem

In this section we prove (5) and (6), that is,

$$V_{m:n} = \frac{m}{n}\left(\frac{n-m}{n-1}\Sigma V + \frac{m-1}{n-1} V\Sigma\right) \quad \text{and} \quad W_p = p\left((1-p)\Sigma V + pV\Sigma\right).$$

By the definitions of variance and covariance, for any subset $I \subseteq [n]$,

$$\mathrm{Var}[\hat{w}_I] = A_I + B_I \quad \text{where} \quad A_I = \sum_{i\in I} \mathrm{Var}[\hat{w}_i] \quad \text{and} \quad B_I = \sum_{i,j\in I, i\neq j} \mathrm{CoV}[\hat{w}_i, \hat{w}_j].$$

Suppose I is chosen uniformly at random among subsets of $[n]$ with m element. Then for any i, $\Pr[i \in I] = m/n$, so by linearity of expectation,

$$\mathrm{E}[A_I] = \sum_{i\in[n]} \Pr[i \in I] \operatorname{Var}[\hat{w}_i] = m/n \cdot A_{[n]}.$$

Also, for any $j \neq i$, $\Pr[i, j \in I] = m/n \cdot (m-1)/(n-1)$, so by linearity of expectation,

$$\mathrm{E}[B_I] = \sum_{i,j\in[n], i\neq j} \Pr[i, j \in I] \operatorname{CoV}[\hat{w}_i, \hat{w}_j] = m/n \cdot (m-1)/(n-1) \cdot B_{[n]}.$$

Thus

$$\mathrm{E}[\operatorname{Var}[\hat{w}_I]] = m/n \cdot A_{[n]} + m/n \cdot (m-1)/(n-1) \cdot B_{[n]} \tag{13}$$

By definition, $A_{[n]} = \Sigma V$. Moreover, by (13), $V\Sigma = A_{[n]} + B_{[n]}$ so $B_{[n]} = V\Sigma - \Sigma V$. Consequently,

$$\begin{aligned} V_{m:n} = \mathrm{E}[\operatorname{Var}[\hat{w}_I]] &= \frac{m}{n}\,\Sigma V + \frac{m}{n}\frac{m-1}{n-1}(V\Sigma - \Sigma V) \\ &= \frac{m}{n}\left(\frac{n-m}{n-1}\,\Sigma V + \frac{m-1}{n-1}\,V\Sigma\right). \end{aligned}$$

This completes the proof of (5).

The proof of (6) is very similar. In this case, each $i \in [n]$ is picked independently for I' with probability p. By linearity of expectation, $\mathrm{E}[A_I] = pA_{[n]}$. Also, for any $j \neq i$, $\Pr[i, j \in I] = p^2$, so by linearity of expectation, $\mathrm{E}[B_I] = p^2 B_{[n]}$. Thus

$$W_p = \mathrm{E}[\operatorname{Var}[\hat{w}_I]] = pA_{[n]} + p^2 B_{[n]} = p\Sigma V + p^2(V\Sigma - \Sigma V) = p((1-p)\Sigma V + pV\Sigma).$$

This completes the proof of (6), hence of Theorem 1.

3 Near-Optimal Schemes

We will now use Theorem 1 to study the average variance (near) optimality of subset sum estimation with threshold sampling, systematic threshold sampling, and priority sampling for any possible set of input weights. The results are all derived based on existing knowledge on ΣV and $V\Sigma$. Below we will focus on $V_{m:n}$ based on random subsets of a given size m. The calculations are very similar for W_p based on the inclusion probability p.

It is well-known from survey sampling that [12, pp. 88,96,97] that systematic sampling always provides an exact estimate of the total so $V\Sigma^{\mathrm{SYS}_k} = 0$. Since variances cannot be negative, we have $V\Sigma^{\mathrm{SYS}_k} = 0 = V\Sigma^{\mathrm{OPT}_k}$. It is also known from survey sampling [12, p. 86] that threshold sampling minimize $V\Sigma$ among all Poisson sampling schemes. In [11] it is further argued that threshold sampling minimizes ΣV over all possible sampling schemes, that

is, $\Sigma V^{\mathrm{THR}_k} = \Sigma V^{\mathrm{OPT}_k}$. Since systematic threshold sampling uses the same marginal distribution for the items, we have $\Sigma V^{\mathrm{THR}_k} = \Sigma V^{\mathrm{SYS}_k} = \Sigma V^{\mathrm{OPT}_k}$. Since SYS_k optimizes both ΣV and $V\Sigma$ we conclude (5) that it optimizes $V_{m:n}$ for any subset size m. More precisely, using (5), we get

$$\begin{aligned} V_{m:n}^{\mathrm{SYS}_k} &= \frac{m}{n}\left(\frac{n-m}{n-1}\,V_1^{\mathrm{SYS}_k} + \frac{m-1}{n-1}\,V_n^{\mathrm{SYS}_k}\right) \\ &= \frac{m}{n}\left(\frac{n-m}{n-1}\,\Sigma V^{\mathrm{OPT}_k} + \frac{m-1}{n-1}\cdot 0\right) \\ &\le V_{m:n}^{\mathrm{OPT}_k} \;\le\; V_{m:n}^{\mathrm{SYS}_k}. \end{aligned}$$

Hence $V_{m:n}^{\mathrm{SYS}_k} = V_{m:n}^{\mathrm{OPT}_k}$.

As mentioned above we have $\Sigma V^{\mathrm{THR}_k} = \Sigma V^{\mathrm{SYS}_k}$. Moreover, threshold sampling has no covariance between individual estimates, so $V_{m:n}^{\mathrm{THR}_k} = \frac{m}{n}\,\Sigma V^{\mathrm{THR}_k} = \frac{m}{n}\,\Sigma V^{\mathrm{OPT}_k}$. But in the previous calculation, we saw that $V_{m:n}^{\mathrm{SYS}_k} = \frac{m}{n}\frac{n-m}{n-1}\,\Sigma V^{\mathrm{OPT}_k}$. Hence we conclude that $V_{m:n}^{\mathrm{SYS}_k} = \frac{n-m}{n-1}\,V_{m:n}^{\mathrm{THR}_k}$. This completes the proof of (8). A very similar calculation establishes (9).

In [13] it is proved that $\Sigma V^{\mathrm{PRI}_{k+1}} \le \Sigma V^{\mathrm{THR}_k} \le \Sigma V^{\mathrm{PRI}_k}$. Moreover, for any scheme Φ without covariance, we have $V_{m:n}^{\Phi} = \frac{m}{n}\,\Sigma V^{\Phi}$. Since both threshold and priority sampling have no covariance, we conclude $V_{m:n}^{\mathrm{PRI}_{k+1}} \le V_{m:n}^{\mathrm{THR}_k} \le V_{m:n}^{\mathrm{PRI}_k}$, which is the statement of (10). The proof of (11) is equivalently based on $W_p^{\Phi} = p\,\Sigma V^{\Phi}$.

4 Extensions

Estimating a Different Variable. Sometimes we wish to estimate a variable u_i different from the weight w_i used for the sampling decision. As an unbiased estimator for u_i, we use $\hat{u}_i = u_i\hat{w}_i/w_i$. We note that Theorem 1 also holds for the variances of the $\hat{u}_i$, but the quality of the different sampling schemes now depend on how well the w_i correlate with the u_i.

The above is the typical model in survey sampling [12], e.g., with u_i being household income and w_i being an approximation of u_i based on known street addresses. In survey sampling, the focus is normally on estimating the total, which is why the parameter $V\Sigma$ is commonly studied. If a sampling scheme has a good $V\Sigma$ w.r.t. $\hat{w}_i$, and w_i is a good approximation of u_i, then it is hoped that $V\Sigma$ is also good w.r.t. $\hat{u}_i$.

The theory developed in this paper provides a corresponding understanding of domain sampling, which is the name for subset sum sampling in survey sampling. To the best of our knowledge, domain sampling was not understood from this perspective.

Biased Estimators. So far we have restricted our attention to unbiased estimators which are normally preferred. We note that Theorem 1 generalizes easily to

biased estimators. Considering the mean square error (MSE) instead of just variance (V), we get the following generalization of (5):

$$MSE_{m:n} = \frac{m}{n}\left(\frac{n-m}{n-1}\,\Sigma MSE + \frac{m-1}{n-1}\,MSE\Sigma\right) \tag{14}$$

Our optimality results for known sampling schemes are, however, restricted to unbiased schemes. Allowing biased estimation, the scheme that minimizes $MSE_{m:n}$ just picks the k largest weights and assigns them some fixed estimates depending on m. This minimizes errors but is totally biased.

References

1. Olken, F., Rotem, D.: Random sampling from databases: a survey. Statistics and Computing 5(1), 25–42 (1995)
2. Haas, P.J.: Speeding up db2 udb using sampling, http://www.almaden.ibm.com/cs/people/peterh/idugjbig.pdf
3. FAQ, O.U.C.O.: http://www.jlcomp.demon.co.uk/faq/random.html
4. Burleson, D.K.: Inside oracle10g dynamic sampling http://www.dba-oracle.com/art_dbazine_oracle10g_dynamic_sampling_hint.htm
5. Alon, N., Duffield, N.G., Lund, C., Thorup, M.: Estimating arbitrary subset sums with few probes. In: Proc. 24th PODS, pp. 317–325 (2005)
6. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. In: Proc. ACM SIGMOD, pp. 171–182. ACM Press, New York (1997)
7. Johnson, T., Muthukrishnan, S., Rozenbaum, I.: Sampling algorithms in a stream operator. In: Proc. ACM SIGMOD, pp. 1–12. ACM Press, New York (2005)
8. Garofalakis, M.N., Gibbons, P.B.: Approximate query processing: Taming the terabytes. In: Proc. 27th VLDB, Tutorial 4 (2001)
9. Duffield, N.G., Lund, C., Thorup, M.: Learn more, sample less: control of volume and variance in network measurements. IEEE Transactions on Information Theory 51(5), 1756–1775 (2005)
10. Cohen, E.: Size-estimation framework with applications to transitive closure and reachability. J. Comput. Syst. Sci. 55(3), 441–453 (1997)
11. Duffield, N.G., Lund, C., Thorup, M.: Sampling to estimate arbitrary subset sums. Technical Report cs.DS/0509026, Computing Research Repository (CoRR), Preliminary journal version of [16] (2005)
12. Särndal, C., Swensson, B., Wretman, J.: Model Assisted Survey Sampling. Springer, Heidelberg (1992)
13. Szegedy, M.: The DLT priority sampling is essentially optimal. In: Proc. 38th ACM Symp. Theory of Computing (STOC), pp. 150–158. ACM Press, New York (2006)
14. Adler, R., Feldman, R., Taqqu, M.: A Practical Guide to Heavy Tails. Birkhauser (1998)
15. Park, K., Kim, G., Crovella, M.: On the relationship between file sizes, transport protocols, and self-similar network traffic. In: Proc. 4th IEEE Int. Conf. Network Protocols (ICNP), IEEE Computer Society Press, Los Alamitos (1996)
16. Duffield, N.G., Lund, C., Thorup, M.: Flow sampling under hard resource constraints. In: Proc. ACM IFIP Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance), pp. 85–96. ACM Press, New York (2004)

On Minimum Power Connectivity Problems*

Yuval Lando and Zeev Nutov

The Open University of Israel, Raanana, Israel
ylando@hotmail.com, nutov@openu.ac.il

Abstract. Given a (directed or undirected) graph with costs on the edges, the power of a node is the maximum cost of an edge leaving it, and the power of the graph is the sum of the powers of its nodes. Motivated by applications for wireless networks, we present improved approximation algorithms and inapproximability results for some classic network design problems under the power minimization criteria. In particular, we give a logarithmic approximation algorithm for the problem of finding a min-power subgraph that contains k internally-disjoint paths from a given node s to every other node, and show that several other problems are unlikely to admit a polylogarithmic approximation.

1 Introduction

1.1 Preliminaries

A large research effort focused on designing "cheap" networks that satisfy prescribed requirements. In wired networks, the goal is to find a subgraph of the minimum cost. In wireless networks, a range (power) of the transmitters determines the resulting communication network. We consider finding a power assignment to the nodes of a network such that the resulting communication network satisfies prescribed connectivity properties and the total power is minimized. Node-connectivity is more central here than edge-connectivity, as it models stations crashes. For motivation and applications to wireless networks (which is the same as of their min-cost variant for wired networks), see, e.g., [15,1,3,16,19].

Let $G = (V, E)$ be a (possibly directed) graph with edge costs $\{c(e) : e \in E\}$. For $v \in V$, the *power* $p(v) = p_c(v)$ *of* v in G (w.r.t. c) is the maximum cost of an edge leaving v in G (or zero, if no such edge exists). The power $p(G) = \sum_{v \in V} p(v)$ of G is the sum of powers of its nodes. Note that $p(G)$ differs from the ordinary cost $c(G) = \sum_{e \in E} c(e)$ of G even for unit costs; for unit costs, if G is undirected then $c(G) = |E|$ and $p(G) = |V|$. For example, if E is a perfect matching on V then $p(G) = 2c(G)$. If G is a clique then $p(G)$ is roughly $c(G)/\sqrt{|E|/2}$. For directed graphs, the ratio of cost over the power can be equal to the maximum outdegree of a node in G, e.g., for stars with unit costs. The following statement shows that these are the extremal cases for general costs.

* This research was supported by The Open University of Israel's Research Fund, grant no. 46102.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 87–98, 2007.

Proposition 1 ([16]). *$c(G)/\sqrt{|E|/2} \leq p(G) \leq 2c(G)$ for any undirected graph $G = (V, E)$, and $c(G) \leq p(G) \leq 2c(G)$ if G is a forest. If G is directed then $c(G)/d_{\max} \leq p(G) \leq c(G)$, $d_{\max}$ is the maximum outdegree of a node in G.*

Simple connectivity requirements are: "st-path for a given node pair s, t", and "path from s to any other node". Min-cost variants are the (directed/undirected) Shortest Path problem and the Min-Cost Spanning Tree problem. In the min-power case, the directed/undirected Min-Power st-Path problem is solvable in polynomial time by a simple reduction to the min-cost case. The undirected Min-Power Spanning Tree problem is APX-hard and admits a $(5/3+\varepsilon)$-approximation algorithm [1]. The directed case is at least as hard as the Set-Cover problem, and thus has an $\Omega(\log n)$-approximation threshold; the problem also admits an $O(\ln n)$-approximation algorithm [3,4]. However, the "reverse" directed min-power spanning tree problem, when we require a path from every node to s, is equivalent to the min-cost case, and thus is solvable in polynomial time.

1.2 Problems Considered

An important network property is fault-tolerance. An edge set E on V is a *k-(edge-)cover* (of V) if the degree (the indegree, in the case of directed graphs) of every node $v \in V$ w.r.t. E is at least k. A graph G is *k-outconnected from s* if it has k (pairwise) internally disjoint sv-paths for any $v \in V$; G is *k-inconnected to s* if its reverse graph is k-outconnected from s (for undirected graphs these two concepts are the same); G is *k-connected* if it has k internally disjoint uv-paths for all $u, v \in V$. When the paths are required only to be edge-disjoint, the graph is *k-edge outconnected from s*, *k-edge inconnected to s*, and *k-edge-connected*, respectively (for undirected graphs these three concepts are the same).

We consider the following classic problems in the power model, some of them generalizations of the problems from [1,3,4], that were already studied, c.f., [16,19,23]. These problems are defined for both directed and undirected graphs.

Min-Power k-Edge-Cover (MP-k-EMC)
Instance: A graph $\mathcal{G} = (V, \mathcal{E})$, edge-costs $\{c(e) : e \in \mathcal{E}\}$, an integer k.
Objective: Find a min-power k-edge-cover $E \subseteq \mathcal{E}$.

Min-Power k Disjoint Paths (MPk-DP)
Instance: A graph $\mathcal{G} = (V, \mathcal{E})$, edge-costs $\{c(e) : e \in \mathcal{E}\}$, $s, t \in V$, an integer k.
Objective: Find a min-power subgraph G of $\mathcal{G}$ with k internally-disjoint st-paths.

Min-Power k-Inconnected Subgraph (MPk-IS)
Instance: A graph $\mathcal{G} = (V, \mathcal{E})$, edge-costs $\{c(e) : e \in \mathcal{E}\}$, $s \in V$, an integer k.
Objective: Find a min-power k-inconnected to s spanning subgraph G of $\mathcal{G}$.

Min-Power k-Outconnected Subgraph (MPk-OS)
Instance: A graph $\mathcal{G} = (V, \mathcal{E})$, edge-costs $\{c(e) : e \in \mathcal{E}\}$, $s \in V$, an integer k.
Objective: Find a min-power k-outconnected from s spanning subgraph G of $\mathcal{G}$.

Min-Power k-Connected Subgraph (MPk-CS)
Instance: A graph $\mathcal{G} = (V, \mathcal{E})$, edge-costs $\{c(e) : e \in \mathcal{E}\}$, an integer k.
Objective: Find a min-power k-connected spanning subgraph G of $\mathcal{G}$.

When the paths are required only to be *edge-disjoint* we get the problems:

Min-Power k Edge-Disjoint Paths (MPk-EDP) (instead of MPk-DP);
Min-Power k-Edge-Inconnected Subgraph (MPk-EIS) (instead of MPk-IS);
Min-Power k-Edge-Outconnected Subgraph (MPk-EOS) (instead of MPk-OS);
Min-Power k-Edge-Connected Subgraph (MPk-ECS) (instead of MPk-CS).

Note that for undirected graphs, the three edge-connectivity problems MPk-EIS, MPk-EOS, and MPk-ECS, are equivalent, but none of them is known to be equivalent to the other for directed graphs. For node connectivity and undirected graphs, only MPk-IS and MPk-OS are equivalent.

1.3 Previous Work

Min-cost versions of the above problems were studied extensively, see, e.g., [9,13,14,12], and surveys in [7,11,17,22].

Previous results on MP-k-EMC: The min-cost variant of MP-k-EMC is essentially the classic b-matching problem, which is solvable in polynomial time, c.f., [7]. The min-power variant MP-k-EMC was shown recently in [19] to admit a $\min\{k+1, O(\ln n)\}$-approximation algorithm, improving the $\min\{k+1, O(\ln^4 n)\}$-approximation achieved in [16]. For directed graphs, MP-k-EMC admits an $O(\ln n)$-approximation algorithm, and this is tight, see [19].

Previous results on MPk-DP: The min cost variant of directed/undirected MPk-EDP/MPk-DP is polynomially solvable, as this is the classic (incapacitated) Min-Cost k-Flow problem, c.f., [7]. In the min-power case, the edge-connectivity variant is substantially harder than the node-connectivity one. In the node-connectivity case, directed MPk-DP is solvable in polynomial time by a simple reduction to the min-cost case, c.f., [16]; this implies a 2-approximation algorithm for undirected MPk-DP, which is however not known to be in P nor NP-hard. In the edge-connectivity case, directed MPk-EDP cannot be approximated within $O(2^{\log^{1-\varepsilon} n})$ for any fixed $\varepsilon > 0$, unless NP $\subseteq$ DTIME$(n^{\text{polylog}(n)})$ [16]. The best known approximation algorithm for directed MPk-EDP is k [23].

Previous results on MPk-IS and MPk-OS: For directed graphs the min-cost versions of MPk-EOS and MPk-OS are polynomially solvable, see [9] and [13,12], respectively. This implies a 2-approximation algorithm for undirected graphs. In the min-power case, the best known approximation ratio for directed MPk-IS/MPk-EIS is k, and the best known ratio for directed MPk-OS/MPk-EOS is $O(k \ln n)$, see [23]. For undirected graphs, the previously best ratio was $2k - 1/3$ for both edge and node-connectivity versions of MPk-IS and MPk-OS, see [19].

Previous results on MPk-CS: Min-cost versions of MPk-CS/MPk-ECS were extensively studied, see surveys in [17] and [22]. The best known approximation ratios for the min-cost variant of MPk-CS are $O(\ln^2 k \cdot \min\{\frac{n}{n-k}, \frac{\sqrt{k}}{\ln k}\})$ for both directed and undirected graphs [21], and $O(\ln k)$ for undirected graphs with $n \geq 2k^2$ [6]. For the edge connectivity variant, there is a simple 2-approximation for both directed and undirected graphs [18]. For the min-power case, the best known

ratio for undirected MPk-CS is $O(\alpha+\ln n)$ [19], where α is the best known ratio for the min-cost case. This result relies on the $O(\ln n)$-approximation for undirected MP-k-EMC of [19], and the observation from [16] that an α-approximation algorithm for the min-cost variant of MPk-CS and a β-approximation algorithm for MP-k-EMC implies a $(2\alpha+\beta)$-approximation algorithm for MPk-CS. For the edge-connectivity variant, the best known ratios are: $2k-1/3$ for undirected graphs [19], and $O(k\ln k)$ for directed graphs [23].

1.4 Results in This Paper

Our first result is for undirected node-connectivity problems. We show that MPk-CS/MPk-OS cannot achieve a better approximation ratio than MP-$(k-1)$-EMC. We also show that up to constants, MPk-OS and MP-k-EMC are equivalent w.r.t. approximation. We use this to get the first polylogarithmic approximation for MPk-OS, improving the ratio $2k-1/3$ by [19]. Formally:

Theorem 1

(i) *If there exists a ρ-approximation algorithm for undirected* MPk-OS/MPk-CS *then there exist a ρ-approximation algorithm for* MP-$(k-1)$-EMC.
(ii) *If there exist a β-approximation algorithm for* MP-$(k-1)$-EMC, *then there exists a $(\beta+4)$-approximation algorithm for undirected* MPk-OS.
(iii) *Undirected* MPk-OS *admits a* $\min\{k+4, O(\log n)\}$*-approximation algorithm.*

Our second result is for undirected MPk-EDP and MPk-ECS. As was mentioned, for $k=1$ directed/undirected MPk-EDP is easily reduced to the min-cost case. We are not aware of any nontrivial algorithms for undirected MPk-EDP for arbitrary k. We give a strong evidence that a polylogarithmic approximation algorithm for undirected MPk-EDP/MPk-ECS may not exist even for highly restricted instances. For that, we show a reduction from the following extensively studied problem to the undirected MPk-EDP/MPk-ECS. For a graph $\mathcal{J}=(V,\mathcal{I})$ and $X\subseteq V$ let $\mathcal{I}(X)$ denote the edges in $\mathcal{I}$ with both ends in X.

Densest ℓ-Subgraph (Dℓ-S)
Instance: A graph $\mathcal{J}=(V,\mathcal{I})$ and an integer ℓ.
Objective: Find $X\subseteq V$ with $|X|\leq\ell$ and $|\mathcal{I}(X)|$ maximum.

The best known approximation ratio for Dℓ-S is roughly $|V|^{-1/3}$ [10] even for the case of bipartite graphs (which up to a factor of 2 is as hard to approximate as the general case), and this ratio holds for 10 years.

We also consider the following "augmentation" version of undirected MPk-EDP (the directed case is easy, c.f., [23]), which already generalizes the case $k=1$ considered in [1].

Min-Power k Edge-Disjoint Paths Augmentation (MPk-EDPA)
Instance: A graph $\mathcal{G}=(V,\mathcal{E})$, edge-costs $\{c(e): e\in\mathcal{E}\}$, $s,t\in V$, an integer k, and a subgraph $G_0=(V,E_0)$ of $\mathcal{G}$ that contains $k-1$ pairwise edge-disjoint st-paths.

Objective: Find $F \subseteq \mathcal{E} - E_0$ so that $G_0 + F$ contains k pairwise edge-disjoint st-paths and with $p(G_0 + F) - p(G_0)$ minimum.

Theorem 2

(i) *Undirected* MPk-EDP/MPk-ECS *admit no* $C \ln n$*-approximation algorithm for some universal constant* $C > 0$*, unless* $P = NP$.

(ii) *If there exists a* ρ*-approximation algorithm for undirected* MPk-EDP *or to undirected* MPk-ECS*, then there exists a* $1/(2\rho^2)$*-approximation algorithm for* Dℓ-S *on bipartite graphs.*

(iii) *Undirected* MPk-EDPA *is in* P*; thus undirected* MPk-EDP *admits a* k*-approximation algorithm.*

Our last result is for edge-connectivity directed problems. In [23] is given an $O(k \ln n)$-approximation algorithm for directed MPk-OS, MPk-EOS, and MPk-ECS, and a k-approximation algorithm for directed MPk-EIS; these ratios are tight up to constant factor if k is "small", but may seem weak if k is large. We prove that for each one of these four problems a polylogarithmic approximation ratio is unlikely to exist even when the costs are symmetric.

Theorem 3. *Directed* MPk-EDP/MPk-EOS/MPk-EIS/MPk-ECS *cannot be approximated within* $O(2^{\log^{1-\varepsilon} n})$ *for any fixed* $\varepsilon > 0$ *even for symmetric costs, unless* NP $\subseteq$ DTIME$(n^{\text{polylog}(n)})$.

Theorems 1, 2, and 3, are proved in Sections 2, 3, and 4, respectively.

Table 1 summarizes the currently best known approximation ratios and thresholds for the connectivity problems considered. Our results show that each one of the directed/undirected edge-connectivity problems MPk-EDP, MPk-EOS, MPk-EIS, MPk-ECS, is unlikely to admit a polylogarithmic approximation. Note again that *directed* MPk-EOS and MPk-EIS are *not* equivalent.

Table 1. Currently best known approximation ratios and thresholds for min-power connectivity problems. Results without references are proved in this paper. σ is the best ratio for Dℓ-S; currently σ is roughly $O(n^{-1/3})$ [10]. β is the best ratio for MP-$(k-1)$-EMC; currently $\beta = \min\{k, O(\log n)\}$ [19]. α is the best ratio for the Min-Cost k-Connected Subgraph problem; currently, $\alpha = \lceil (k+1)/2 \rceil$ for $2 \le k \le 7$ (see [2] for $k = 2, 3$, [8] for $k = 4, 5$, and [20] for $k = 6, 7$); $\alpha = k$ for $k = O(\ln n)$ [20], $\alpha = 6H(k)$ for $n \ge k(2k-1)$ [6], and $\alpha = O(\ln k \cdot \min\{\sqrt{k}, \frac{n}{n-k} \ln k\})$ for $n < k(2k-1)$ [21].

Problem	Edge-Connectivity		Node-Connectivity	
	Undirected	*Directed*	*Undirected*	*Directed*
MPk-DP	k	k [23]	2 [16]	in P [16]
	$\Omega(\max\{1/\sqrt{\sigma}, \ln n\})$	$\Omega(2^{\log^{1-\varepsilon} n})$ [16]	$--$	
MPk-IS	$2k - 1/3$ [19]	k [23]	$\min\{k+4, O(\ln n)\}$	k [23]
	$\Omega(\max\{1/\sqrt{\sigma}, \ln n\})$	$\Omega(2^{\log^{1-\varepsilon} n})$	$\Omega(\beta)$	$--$
MPk-OS	$2k - 1/3$ [19]	$O(k \ln n)$ [23]	$\min\{k+4, O(\ln n)\}$	$O(k \ln n)$ [23]
	$\Omega(\max\{1/\sqrt{\sigma}, \ln n\})$	$\Omega(2^{\log^{1-\varepsilon} n})$	$\Omega(\beta)$	$\Omega(\ln n)$ for $k = 1$ [3]
MPk-CS	$2k - 1/3$ [19]	$O(k \ln n)$ [23]	$O(\alpha + \ln n)$ [19]	$O(k(\ln n + k))$ [23]
	$\Omega(\max\{1/\sqrt{\sigma}, \ln n\})$	$\Omega(2^{\log^{1-\varepsilon} n})$	$\Omega(\beta)$	$\Omega(\ln n)$ for $k = 1$ [3]

1.5 Notation

Here is some notation used in the paper. Let $G = (V, E)$ be a (possibly directed) graph. Let $\deg_E(v) = \deg_G(v)$ denote the degree of v in G. Given edge costs $\{c(e) : e \in E\}$, the power of v in G is $p_c(v) = p(v) = \max_{vu \in E} c(e)$, and the power of G is $p(G) = p(V) = \sum_{v \in V} p(v)$. Throughout the paper, $\mathcal{G} = (V, \mathcal{E})$ denotes the input graph with nonnegative costs on the edges. Let $n = |V|$ and $m = |\mathcal{E}|$. Given $\mathcal{G}$, our goal is to find a minimum power spanning subgraph $G = (V, E)$ of $\mathcal{G}$ that satisfies some prescribed property. We assume that a feasible solution exists; let opt denote the optimal solution value of an instance at hand.

2 Proof of Theorem 1

In this section we consider undirected graphs only and prove Theorem 1. Part (iii) of Theorem 1 follows from Part (ii) and the fact that MP-$(k-1)$-EMC admits a $\min\{k, O(\log n)\}$-approximation algorithm [19]. In the rest of this section we prove Parts (i) and (ii) of Theorem 1.

We start by proving Part (i). The reduction for MPk-CS is a s follows. Let $\mathcal{G} = (V, \mathcal{E}), c$ be an instance of MP-$(k-1)$-EMC with $|V| \geq k$. Construct an instance $\mathcal{G}', c'$ for MPk-CS as follows. Add a copy V' of V and the set of edges $\{vv' : v \in V\}$ of cost 0 ($v' \in V'$ is the copy of $v \in V$), and then add a clique of cost 0 on V'. Let $\mathcal{E}'$ be the edges of $\mathcal{G}' - \mathcal{E}$. We claim that $E \subseteq \mathcal{E}$ is a $(k-1)$-edge cover if, and only if, $G' = (V + V', E + \mathcal{E}')$ is k-connected.

Suppose that G' is k-connected. Then $\deg_{E+\mathcal{E}'}(v) \geq k$ and $\deg_{E'}(v) = 1$ for all $v \in V$. Hence $\deg_{\mathcal{E}}(v) \geq k-1$ for all $v \in V$, and thus E is a $(k-1)$-edge-cover.

Suppose that $E \subseteq \mathcal{E}$ is a $(k-1)$-edge cover. We will show that G' has k internally disjoint vu-paths for any $u, v \in V + V'$. It is clear that $G' - E$, and thus also G', has k internally disjoint vu-paths for any $u, v \in V'$. Let $v \in V$. Consider two cases: $u \in V'$ and $u \in V$. Assume that $u \in V'$. Every neighbor v_i of v in (V, E) defines the vu path (v, v_i, v_i', u) (possibly $v_i' = u$), which gives $\deg_E(v) \geq k-1$ internally disjoint vu-paths. An additional path is (v, v', u). Now assume that $u \in V$. Every common neighbor a of u and v defines the vu-path (v, a, u), and suppose that there are q such common neighbors. Each of v and u has at least $k-1-q$ more neighbors in G, say $\{v_1, \ldots, v_{k-1-q}\}$ and $u_1, \ldots, u_{k-1-q}$, respectively. This gives $k-1-q$ internally disjoint vu-paths (v, v_i, v_i', u_i', u), $i = 1, \ldots, k-1-q$. An additional path is (v, v', u', u). It is easy to see that these k vu-paths are internally disjoint. The proof for MPk-CS is complete.

The reduction for MPk-OS is the same, except that in the construction of $\mathcal{G}'$ we also add a node s and edges $\{sv' : v' \in V'\}$ of cost 0.

We now prove Part (ii); the proof is similar to the proof of Theorem 3 from [16]. Given a graph G which is k-outconnected from s, let us say that an edge e of G is *critical* if $G - e$ is not k-outconnected from s. We need the following fundamental statement:

Theorem 4 ([5]). *In a k-outconnected from s undirected graph G, any cycle of critical edges contains a node $v \neq s$ whose degree in G is exactly k.*

The following corollary (e.g., see [5]) is used to get a relation between $(k-1)$-edge covers and k-outconnected subgraphs.

Corollary 1. *If* $\deg_J(v) \geq k-1$ *for every node* v *of an undirected graph* J*, and if* F *is an inclusion minimal edge set such that* $J \cup F$ *is* k*-outconnected from* s*, then* F *is a forest.*

Proof. If not, then F contains a critical cycle C, but every node of C is incident to 2 edges of C and to at least $k-1$ edges of J, contradicting Theorem 4. □

We now finish the proof of Part (ii). By the assumption, we can find a subgraph J with $\deg_J(v) \geq k-1$ of power at most $p(J) \leq \rho\mathsf{opt}$. We reset the costs of edges in J to zero, and apply a 2-approximation algorithm for the Min-Cost k-Outconnected Subgraph problem (c.f., [13]) to compute an (inclusion) minimal edge set F so that $J+F$ is k-outconnected from s. By Corollary 1, F is a forest. Thus $p(F) \leq 2c(F) \leq 4\mathsf{opt}$, by Proposition 1. Combining, we get Part (ii).

3 Proof of Theorem 2

It is easy to see that MPk-EDP is "Set-Cover hard". Indeed, the Set-Cover problem can be formulated as follows. Given a bipartite graph $J = (A+B, E)$, find a minimum size subset $S \subseteq A$ such that every node in B has a neighbor in S. Construct an instance of MPk-EDP by adding two new nodes s and t, edges $\{sa : a \in A\} \cup \{bt : b \in B\}$, and setting $c(e) = 1$ if e is incident to s and $c(e) = 0$ otherwise. Then replace every edge not incident to t by $|B|$ parallel edges. For $k = |B|$, it is easy to see that S is a feasible solution to the Set-Cover instance if, and only if, the subgraph induced by $S \cup B \cup \{s,t\}$ is a feasible solution of power $|S|+1$ to the obtained MPk-EDP instance; each node in $S \cup \{t\}$ contributes 1 to the total power, while the contribution of any other node is 0. Combining this with the hardness results of [25] for Set-Cover, we get Part (i).

The proof of Part (ii) is similar to the the proof of Theorem 1.2 from [24], where the related Node-Weighted Steiner Network was considered; we provide the details for completeness of exposition. We need the following known statement (c.f., [10,24]).

Lemma 1. *There exists a polynomial time algorithm that given a graph* $G = (V, E)$ *and an integer* $1 \leq \ell \leq n = |V|$ *finds a subgraph* $G' = (V', E')$ *of* G *with* $|V'| = \ell$ *and* $|E'| \geq |E| \cdot \frac{\ell(\ell-1)}{n(n-1)}$.

Given an instance $\mathcal{J} = (A+B, \mathcal{I})$ and ℓ of bipartite Dℓ-S, define an instance of (undirected) unit-cost MPk-EDP/MPk-ECS by adding new nodes $\{s,t\}$, a set of edges $E_0 = \{aa' : a, a' \in A+s\} \cup \{bb' : b, b' \in B+t\}$ of multiplicity $|A|+|B|$ and cost 0 each, and setting $c(e) = 1$ for all $e \in \mathcal{I}$. It is easy to see that any $E \subseteq \mathcal{I}$ determines $|E|$ edge-disjoint st-paths, and that $(A+B+\{s,t\}, E_0+|E|)$ is k-connected if, and only if, $|E| \geq k$. Thus for any integer $k \in \{1, \ldots, |\mathcal{I}|\}$, if we have a ρ-approximation algorithm for undirected MPk-EDP/MPk-ECS, then

we have a ρ-approximation algorithm for $\min\{|X| : X \subseteq A+B, |\mathcal{I}(X)| \geq k\}$. We show that this implies a $1/(2\rho^2)$-approximation algorithm for the original instance of bipartite Dℓ-S, which is $\max\{|\mathcal{I}(X)| : X \subseteq A+B, |X| \leq \ell\})$.

For every $k = 1, \ldots, |\mathcal{I}|$, use the ρ-approximation algorithm for undirected MPk-EDP/MPk-ECS to compute a subset $X_k \subseteq A+B$ so that $|\mathcal{I}(X_k)| \geq k$, or to determine that no such X_k exists. Set $X = X_k$ where k is the largest integer so that $|X_k| \leq \min\{\lfloor \rho \cdot \ell \rfloor, |A| + |B|\}$ and $|\mathcal{I}(X_k)| \geq k$. Let X^* be an optimal solution for Dℓ-S. Note that $|\mathcal{I}(X)| \geq |\mathcal{I}(X^*)|$ and that $\frac{\ell(\ell-1)}{|X|(|X|-1)} \geq 1/(2\rho^2)$. By Lemma 1 we can find in polynomial time $X' \subseteq X$ so that $|X'| = \ell$ and $|\mathcal{I}(X')| \geq |\mathcal{I}(X)| \cdot \frac{\ell(\ell-1)}{|X|(|X|-1)} \geq |\mathcal{I}(X^*)| \cdot 1/(2\rho^2)$. Thus X' is a $1/(2\rho^2)$-approximation for the original bipartite Dℓ-S instance.

We now prove part (iii) of Theorem 2. It would be convenient to describe the algorithm using "mixed" graphs that contain both directed and undirected edges. Given such mixed graph with weights on the nodes, a minimum weight path between two given nodes can be found in polynomial time using Dikjstra's algorithm and elementary constructions. The algorithm for undirected MPk-EDPA is as follows.

1. Construct a graph $\mathcal{G}'$ from $\mathcal{G}$ as follows. Let $p_0(v)$ be the power of v in G_0. For every $v \in V$ do the following. Let $p_0(v) \leq c_1 < c_2 < \cdots$ be the costs of the edges in $\mathcal{E}$ leaving v of cost at least $p_0(v)$ sorted in increasing order. For every c_j add a node v_j of the weight $w(v_j) = c_j - p_0(v)$. Then for every $u_{j'}, v_{j''}$ add an edge $u_{j'}v_{j''}$ if $w(u_{j'}), w(v_{j''}) \geq c(uv)$. Finally, add two nodes s, t and an edge from s to every s_j and from every t_j to t.
2. Construct a mixed graph $\mathcal{D}$ from $\mathcal{G}'$ as follows. Let I be an inclusion minimal edge set in G_0 that contains $k-1$ pairwise edge-disjoint st-paths. Direct those paths from t to s, and direct accordingly every edge of $\mathcal{G}'$ that corresponds to an edge in I.
3. In $\mathcal{D}$, compute a minimum weight st-path P. Return the set of edges of $\mathcal{G}$ that correspond to P that are not in E_0.

We now explain why the algorithm is correct. It is known that the following "augmenting path" algorithm solves the **Min-Cost k Edge-Disjoint Paths Augmentation** problem (the min-cost version of MPk-EDPA, where the edges in G_0 have cost 0) in undirected graphs (c.f., [7]).

1. Let I be an inclusion minimal edge set in G_0 that contains $k-1$ pairwise edge-disjoint st-paths. Construct a mixed graph $\mathcal{D}$ from $\mathcal{G}$ by directing these paths from t to s.
2. Find a min-cost path P in $\mathcal{D}$. Return $P - E_0$.

Our algorithm for MPk-EDPA does the same but on the graph $\mathcal{G}'$. The feasibility of the solution follows from standard network flow arguments. The key point in proving optimality is that in $\mathcal{G}'$ the weight of a node is the increase of its power caused by taking an edge incident to this node. It can be shown that for any feasible solution F corresponds a unique path P in $\mathcal{D}$ so that $p(G_0 + F) - p(G_0) = w(P)$, and vice versa. As we choose the minimum weight path in $\mathcal{D}$, the returned solution is optimal.

4 Proof of Theorem 3

4.1 Arbitrary Costs

We first prove Theorem 3 for arbitrary costs, not necessarily symmetric. For that, we use the hardness result for MPk-EDP of [16], to show a similar hardness for the other three problems MPk-EOS, MPk-EIS, and MPk-ECS. Loosely speaking, we show that each of directed MPk-EOS/MPk-EIS is at least as hard as MPk-EDP, and that MPk-ECS is at least as hard as MPk-EOS.

We start by describing how to reduce directed MPk-EDP to directed MPk-EOS. Given an instance $\mathcal{G} = (V, \mathcal{E}), c, (s, t), k$ of MPk-EDP construct an instance of $\mathcal{G}' = (V', \mathcal{E}'), c', s, k$ of directed MPk-EOS as follows. Add to $\mathcal{G}$ a set $U = \{u_1, \ldots, u_k\}$ of k new nodes, and then add an edge set E_0 of cost zero: from t to every node in U, and from every node in U to every node $v \in V - \{s, t\}$. That is

$$V' = V + U = V + \{u_1, \ldots, u_k\},$$
$$\mathcal{E}' = \mathcal{E} + E_0 = \mathcal{E} + \{tu : u \in U\} + \{uv : u \in U, v \in V' - \{s, t\}\},$$
$$c'(e) = c(e) \text{ if } e \in \mathcal{E} \text{ and } c'(e) = 0 \text{ otherwise .}$$

Since in the construction $|V'| = |V| + k \leq |V| + |V|^2 \leq 2|V|^2$, Theorem 2 together with the following claim implies Theorem 3 for asymmetric MPk-EOS.

Claim. $G = (V, E)$ is a solution to the MPk-EDP instance if, and only if, $G' = (V', E' = E + E_0)$ is a solution to the constructed MPk-EOS instance.

Proof. Let E be a solution to the MPk-EDP instance and let $\Pi = \{P_1, \ldots, P_k\}$ be a set of k pairwise edge-disjoint st-paths in E. Then in $G' = (V', E + E_0)$ for every $v \in V' - s$ there is a set $\Pi' = \{P'_1, \ldots, P'_k\}$ of k pairwise edge-disjoint sv-paths: if $v = t$ then $\Pi' = \Pi$; if $v \neq s$ then $P'_j = P_j + tu_j + u_j v$, $j = 1, \ldots, k$.

Now let $E' = E + E_0$ be a solution to constructed MPk-EOS instance. In particular, (V', E') contains a set Π of k-edge disjoint st-paths, none of which has t as an internal node. Consequently, no path in Π passes through U, as t is the tail of every edge entering U. Thus Π is a set k-edge disjoint st-paths in G, namely, $G = (V, E)$ is a solution to the original MPk-EDP instance. □

Asymmetric MPk-EIS*:* The reduction of asymmetric MPk-EIS to MPk-EDP is similar to the one described above, except that here set $E_0 = \{us : u \in U\} + \{vu : v \in V - \{s, t\}, u \in U\}$; namely, connect every $u \in U$ to s, and every $v \in V - \{s, t\}$ to every $u \in U$. Then in the obtained MPk-EIS instance, require k internally edge-disjoint vt-paths for every $v \in V$, namely, we seek a graph that is k-edge-inconnected to t. The other parts of the proof for MPk-EIS are identical to those for MPk-EOS described above.

Asymmetric MPk-ECS*:* Reduce the directed MPk-EOS to the directed MPk-ECS as follows. Let $\mathcal{G} = (V, \mathcal{E}), c, s, k$ be an instance of MPk-EOS. Construct an instance of $\mathcal{G}' = (V', \mathcal{E}'), c', s, k$ of MPk-EOS as follows. Add to $\mathcal{G}$ a set $U = \{u_1, \ldots, u_k\}$ of k new nodes, and then add an edge set $E_0 = \{uu' : u, u' \in U\} + \{vu : v \in V - s, u \in U\} + \{us : u \in U\}$ of cost 0; namely, E_0 is obtained by taking a complete graph on U and adding all edges from $V - s$ to U and all

edges from U to s. It is not hard to verify that if $E \subseteq \mathcal{E}$, then $G = (V, E)$ is k-edge-outconnected from s if, and only if, $G' = (V', E_0 + E)$ is k-edge-connected.

4.2 Symmetric Costs

We show that the directed problems MPk-EDP, MPk-EOS, MPk-EIS, MPk-ECS are hard to approximate even for symmetric costs. We start with directed symmetric MPk-EDP. We use a refinement of a result from [16] which states that directed MPk-EDP cannot be approximated within $O(2^{\log^{1-\varepsilon} n})$ for any fixed $\varepsilon > 0$, unless NP $\subseteq$ DTIME$(n^{\text{polylog}(n)})$. In [16] it is shown that this hardness result holds for simple graphs with costs in $\{0, n^3\}$, where $n = |V|$. If we change the cost of every edge of cost 0 to 1, it will add no more then n^2/n^3 to the total cost of any solution that uses at least one edge of cost n^3. Thus we have:

Corollary 2 ([16]). *Directed* MPk-EDP *with costs in* $\{1, n^3\}$ *cannot be approximated within* $O(2^{\log^{1-\varepsilon} n})$ *for any fixed* $\varepsilon > 0$, *unless* NP $\subseteq$ DTIME$(n^{\text{polylog}(n)})$.

We show that a ρ-approximation algorithm for directed symmetric MPk-EDP implies a ρ-approximation algorithm for directed MPk-EDP with costs in $\{1, n^3\}$, for any $\rho < n^{1/7}$. Let $\mathcal{G} = (V, \mathcal{E}), c, (s, t), k$ be an instance of MPk-EDP with costs in $\{1, n^3\}$. Let opt be an optimal solution value for this instance. Note that opt $\leq n^4$. Let $N = n^5$. Define an instance $\mathcal{G}' = (V', \mathcal{E}'), c', (s, t), k' = kN$ for directed symmetric MPk-EDP as follows. First, obtain $\mathcal{G}^+ = (V', \mathcal{E}^+), c^+$ by replacing every edge $e = uv \in E$ by N internally-disjoint uv-paths of the length 2 each, where the cost of the first edge in each paths is $c(e)$ and the cost of the second edge is 0. Second, to obtain a symmetric instance $\mathcal{G}', c'$, for every edge $ab \in \mathcal{E}^+$ add the opposite edge ba of the same cost as ab.

For a path P^+ in $\mathcal{E}^+$, let $\psi(P^+)$ denote the unique path in $\mathcal{E}$ corresponding to P^+. For any path P in $\mathcal{E}$, the paths in the set $\psi^{-1}(P)$ of the paths in $\mathcal{E}^+$ that corresponds to P are edge-disjoint. Hence, any set Π of paths in $\mathcal{E}$ is mapped by ψ^{-1} to a set $\Pi^+ = \psi^{-1}(\Pi)$ of exactly $N|\Pi|$ edge-disjoint paths in $\mathcal{E}^+$ of the same power, namely $|\Pi^+| = N|\Pi|$ and $p_c(\Pi) = p_{c^+}(\Pi^+)$. Conversely, any set Π^+ of paths in $\mathcal{E}^+$ is mapped by ψ to a set $\Pi = \psi(\Pi^+)$ of at least $\lceil |\Pi^+|/N \rceil$ edge-disjoint paths in $\mathcal{E}$ of the same power, namely, $|\Pi| = \lceil |\Pi|/N \rceil$ and $p_c(\Pi) = p_{c^+}(\Pi^+)$. In particular:

Corollary 3. opt$' \leq$ opt $\leq n^4$, *where* opt$'$ *is an optimal solution value for* $\mathcal{G}'$.

Note that $|V'| = n' \leq n^7$, hence to prove Theorem 3 for directed symmetric MPk-EDP it is sufficient to prove that a $\rho(n')$-approximation algorithm for $\mathcal{G}', c', (s, t), k'$ with $\rho(n') < n'^{1/7}$ implies a $\rho(n)$-approximation algorithm for the original instance. Suppose that we have a $\rho(n')$-approximation algorithm that computes an edge set $E' \subseteq \mathcal{E}'$ that contains a set Π' of kN edge-disjoint paths in $\mathcal{G}'$ of power $p_{c'}(E') \leq \rho \cdot$ opt$'$, where $\rho = \rho(n') < n'^{1/7} \leq n$. Then $|E' - \mathcal{E}^+| \leq \rho \cdot$ opt$' \leq \rho \cdot n^4$, since every edge in $|E' - \mathcal{E}^+|$ adds at least one to $p_{c'}(E')$. Consequently, there is a set $\Pi^+ \subseteq \Pi'$ of at least $kN - \rho \cdot n^4$ paths in Π that are contained in $E^+ = E' \cap \mathcal{E}^+$. Hence, since $\rho = \rho(n') > n'^{1/7} \geq n$, the number of paths in $\Pi = \psi(\Pi^+)$ is at least

$$|\Pi| \geq \left\lceil \frac{kN - \rho \cdot n^4}{N} \right\rceil \geq \left\lceil k - \frac{\rho \cdot n^4}{N} \right\rceil = \left\lceil k - \frac{\rho}{n} \right\rceil \geq k \ .$$

Consequently, the set E of edges of Π is a feasible solution for $\mathcal{G}, c, (s,t), k$ of power at most $p_c(E) \leq p_{c'}(E') \leq \rho \mathsf{opt}' \leq \rho \mathsf{opt}$. Since in the construction $|V'| \leq |V|^7$, Corollary 2 implies Theorem 3 for directed symmetric MPk-EDP.

The proof for the other problems MPk-EOS, MPk-EIS, and MPk-ECS, is similar, with the help of reductions described for the asymmetric case.

5 Conclusion

In this paper we showed that MPk-OS and MP-$(k-1)$-EMC are equivalent w.r.t. approximation, and used this to derive a logarithmic approximation for MPk-OS. We also showed that the edge-connectivity problems MPk-EDP, MPk-EOS/MPk-EIS, and MPk-ECS, are unlikely to admit a polylogarithmic approximation ratio for both directed and undirected graphs, and for directed graphs this is so even if the costs are symmetric. In contrast, we showed that the augmentation version MPk-EDPA of MPk-EDP, can be reduced to the shortest path problem.

We now list some open problems, that follow from Table 1. Most of them concern node-connectivity problems. One open problem is to determine whether the *undirected* MPk-DP is in P or is NP-hard (as was mentioned, the directed MPk-DP is in P, c.f., [16]). In fact, we do not even know whether the augmentation version MPk-DPA of undirected MPk-DP is in P. A polynomial algorithm for undirected MPk-DPA can be used to slightly improve the known ratios for the undirected MPk-CS: from 9 (follows from [20]) to $8\frac{2}{3}$ for $k = 4$, and from 11 (follows from [8]) to 10 for $k = 5$. This is achieved as follows. In [8] it is shown that if G is k-outconnected from r and $\deg_G(r) = k$ then G is $(\lceil k/2 \rceil + 1)$-connected; furthermore, for $k = 4, 5$, G contains two nodes s, t so that increasing the connectivity between them by one results in a k-connected graph. Hence for $k = 4, 5$, we can get approximation ratio $\gamma + \delta$, where γ is the ratio for undirected MPk-OS, and δ is the ratio for undirected MPk-DPA. As MPk-OS can be approximated within $\min\{2k - 1/3, k + 4\}$, then if MPk-DPA is in P, for MPk-CS we can get approximation ratios $8\frac{2}{3}$ for $k = 4$ and 10 for $k = 5$.

Another question is the approximability of the directed MPk-IS. Currently, we are not aware of any hardness result, while the best known approximation ratio is k. Except directed MPk-DP, there is still a large gap between upper and lower bounds of approximation for all the other min-power node connectivity problems, for both directed and undirected graphs.

References

1. Althaus, E., Calinescu, G., Mandoiu, I., Prasad, S., Tchervenski, N., Zelikovsky, A.: Power efficient range assignment for symmetric connectivity in static ad-hoc wireless networks. Wireless Networks 12(3), 287–299 (2006)
2. Auletta, V., Dinitz, Y., Nutov, Z., Parente, D.: A 2-approximation algorithm for finding an optimum 3-vertex-connected spanning subgraph. Journal of Algorithms 32(1), 21–30 (1999)

3. Calinescu, G., Kapoor, S., Olshevsky, A., Zelikovsky, A.: Network lifetime and power assignment in ad hoc wireless networks. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 114–126. Springer, Heidelberg (2003)
4. Caragiannis, I., Kaklamanis, C., Kanellopoulos, P.: Energy-efficient wireless network design. Theory of Computing Systems 39(5), 593–617 (2006)
5. Cheriyan, J., Jordán, T., Nutov, Z.: On rooted node-connectivity problems. Algorithmica 30(3), 353–375 (2001)
6. Cheriyan, J., Vempala, S., Vetta, A.: An approximation algorithm for the minimum-cost k-vertex connected subgraph. SIAM Journal on Computing 32(4), 1050–1055 (2003)
7. Cook, W.J., Cunningham, W.H., Pulleyblank, W.R., Scrijver, A.: Combinatorial Optimization. Wiley, Chichester (1998)
8. Dinitz, Y., Nutov, Z.: A 3-approximation algorithm for finding an optimum 4,5-vertex-connected spanning subgraph. Journal of Algorithms 32(1), 31–40 (1999)
9. Edmonds, J.: Matroid intersection. Annals of discrete Math., 185–204 (1979)
10. Feige, U., Kortsarz, G., Peleg, D.: The dense k-subgraph problem. Algorithmica 29(3), 410–421 (2001)
11. Frank, A.: Connectivity and network flows. In: Graham, R., Grötschel, M., Lovász, L. (eds.) Handbook of Combinatorics, pp. 111–177. Elsvier Science (1995)
12. Frank, A.: Rooted k-connections in digraphs, EGRES TR No 2006-07 (2006)
13. Frank, A., Tardos, E.: An application of submodular flows. Linear Algebra and its Applications 114/115, 329–348 (1989)
14. Gabow, H.N.: A representation for crossing set families with application to submodular flow problems. In: SODA, pp. 202–211 (1993)
15. Hajiaghayi, M.T., Immorlica, N., Mirrokni, V.S.: Power optimization in fault-tolerant topology control algorithms for wireless multi-hop networks. In: Proc. Mobile Computing and Networking (MOBICOM), pp. 300–312 (2003)
16. Hajiaghayi, M.T., Kortsarz, G., Mirrokni, V.S., Nutov, Z.: Power optimization for connectivity problems. Math. Programming 110(1), 195–208 (2007)
17. Khuller, S.: Approximation algorithms for for finding highly connected subgraphs. In: Hochbaum, D.S. (ed.) Approximation Algorithms FOR NP-Hard Problems, ch. 6, pp. 236–265. PWS (1995)
18. Khuller, S., Vishkin, U.: Biconnectivity approximations and graph carvings. Journal of the Association for Computing Machinery 41(2), 214–235 (1994)
19. Kortsarz, G., Mirrokni, V.S., Nutov, Z., Tsanko, E.: Approximation algorithms for minimum power degree and connectivity problems. Manuscript (2006)
20. Kortsarz, G., Nutov, Z.: Approximating node-connectivity problems via set covers. Algorithmica 37, 75–92 (2003)
21. Kortsarz, G., Nutov, Z.: Approximating k-node connected subgraphs via critical graphs. SIAM J. on Computing 35(1), 247–257 (2005)
22. Kortsarz, G., Nutov, Z.: Approximating minimum cost connectivity problems. In: Gonzalez, T.F. (ed.) Approximation Algorithms and Metaheuristics, ch. 58 (2007)
23. Nutov, Z.: Approximating minimum power covers of intersecting families and directed connectivity problems. In: Díaz, J., Jansen, K., Rolim, J.D.P., Zwick, U. (eds.) APPROX 2006 and RANDOM 2006. LNCS, vol. 4110, Springer, Heidelberg (2006)
24. Nutov, Z.: Approximating Steiner networks with node weights. manuscript (2006)
25. Raz, R., Safra, S.: A sub-constant error-probability low-degree test and a sub-constant error-probability PCP characterization of NP. In: STOC, pp. 475–484 (1997)

On the Cost of Interchange Rearrangement in Strings

Amihood Amir[1,2,⋆], Tzvika Hartman[1], Oren Kapah[1], Avivit Levy[1,⋆⋆], and Ely Porat[1]

[1] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
{amir,hartmat,kapaho,levyav2,porately}@cs.biu.ac.il
[2] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218

Abstract. An underlying assumption in the classical sorting problem is that the sorter does not know the index of every element in the sorted array. Thus, comparisons are used to determine the order of elements, while the sorting is done by interchanging elements. In the closely related interchange rearrangement problem, final positions of elements are already given, and the cost of the rearrangement is the cost of the interchanges. This problem was studied only for the limited case of permutation strings, where every element appears once. This paper studies a generalization of the classical and well-studied problem on permutations by considering general strings input, thus solving an open problem of Cayley from 1849, and examining various cost models.

1 Introduction

In the classical sorting problem the input is a list of elements and a relation R, and the goal is to efficiently rearrange the list according to R. The underlying assumption is that the sorter does not know the location of the elements in the sorted list. Therefore, the basic tool used to determine the order of elements is comparisons according to R. The operation used to rearrange elements in the list is an *interchange*, in which the two participating elements exchange positions. Since the number of rearrangements is less than the number of comparisons the cost of a sorting algorithm is usually measured by the number of comparisons it makes. In the closely related *interchange rearrangement* problem the relation R is given implicitly as a target string, representing the desired order between the elements in the list. The goal is to rearrange the string representing the current list to the target string by interchanges. In this problem, however, the underlying assumption is that the sorter can 'see' the whole lists in one operation or get this information for free. Therefore, the sorter does not need comparisons. The cost of the rearrangement is now determined by the cost of the rearrangements operations. Thus, the question is, what is the minimum rearrangement operations required in order to rearrange the input string to the target string. Some

⋆ Partly supported by ISF grant 35/05.
⋆⋆ Partly supported by a BIU President Fellowship. This work is part of A. Levy's Ph.D. thesis.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 99–110, 2007.

situations of objects rearrangement fit the interchange rearrangement problem rather than the classical sorting problem. Consider, for example, a robot rearranging a list of objects (on an assembly line), by interchanging two elements in each operation, one in each 'hand'. The interchanges sequence for the given input is computed in advance.

Rearrangement distances are extensively studied in comparative genomics for some rearrangement operators. Genome rearrangement distances are studied in order to establish phylogenetic information about the evolution of species. During the course of evolution, whole regions of genome may evolve through reversals, transpositions and translocations. Considering the genome as a string over the alphabet of genes, these cases represent a situation where the difference between the original string and the resulting one is in the locations of the different elements, where changes are caused by rearrangements operators. Many works have considered specific versions of this biological setting (*sorting by reversals* [1,2,3], *transpositions* [4], and *translocations* [5]). The biologically meaningful rearrangement operators usually involve blocks in the genome sequence and are not limited to single genes. Thus, although the problem of *sorting by block interchanges* was studied [6], the interchange rearrangement operator on single elements was neglected by genome rearrangements researchers[1]. Nevertheless, we believe that studying this operator, especially in general strings may give insights into other rearrangement operators as well. Most of the work on genome rearrangements is on permutation strings, where every symbol appears only once. The more interesting case of general strings has indeed been dealt by some researchers (e.g. [8,9]), however, still not studied enough.

A first step in the study of the interchange rearrangement problem was made in 1849 by the mathematician Cayley [10], who studied the limited input of permutation strings in which all elements are distinct. In this case strings can be viewed as a permutation of $1, \ldots, m$, where m is the length of the string. This classical setting was well studied. For example, [10,11] give a characteristic theorem for the distance in this case. Recently, [12] defined the *interchange distance problem* and described a simple linear time algorithm for computing it on permutation strings. However, these results do not apply for the general strings case. The general strings case was left as an open problem by Cayley. This paper studies a generalization of this classical and well-studied problem on permutations by considering the general strings input, and examining various cost models.

Rearrangement Cost Models. The popular cost model used in rearrangement distances is the unit cost model. In this model, every rearrangement operation is given a unit cost[2]. However, in many situations there is no reason to assume that all interchanges have equal cost. On the contrary, interchanging closer elements

[1] Interchanges are considered in [7] in the context of genome rearrangements, and are called there *swaps*. However, they only study *short swaps*, in which the distance between the interchanged elements is at most two.

[2] This is the model used in the definition of the interchange distance problem.

would probably be cheaper than interchanging far elements. This motivated us to study the interchange rearrangement under a variety of cost models. In the general cost model every interchange operation has a cost determined by a cost function. Note that every cost function on an interchange of elements in positions i, j where $i < j$, can be viewed as a cost function on the segment $[i, j]$. We are interested in increasing monotone cost functions on the length of the segment, i.e. $|i - j|$. Such cost functions are natural in some real-world situations, where the distance objects have been moved contributes to the cost. Cost functions w of the form $w(\ell) = \ell^\alpha$ for all $\alpha \geq 0$, where $\ell = |i-j|$, are specifically studied. The study is also broadened to include various cost functions (e.g. $\log(\ell)$) classified by their characteristic behavior regarding the marginal cost. Two types of cost functions are considered: the *$\mathcal{I}$-type* and the *$\mathcal{D}$-type* defined as follows[3].

Definition 1. *Let $w : \mathbb{N} \mapsto \mathbb{R}$ a cost function. We say that:*

- *$w \in \mathcal{I}$-type if for every $a, b, c \in \mathbb{N}$ such that $a < b$, $w(a + c) - w(a) < w(b + c) - w(b)$. We call this property* ***the law of increasing marginal cost****.*
- *$w \in \mathcal{D}$-type if for every $a, b, c \in \mathbb{N}$ such that $a < b$, $w(a + c) - w(a) > w(b + c) - w(b)$. We call this property* ***the law of decreasing marginal cost****.*

The Problem Definition. Let $w : \mathbb{N} \mapsto \mathbb{R}$ be a cost function, $x, y \in \Sigma^m$ be two strings, and let $s = s_1, \ldots, s_k$ be a sequence of interchanges that converts x to y, where s_j interchanges elements in positions i_j, i'_j, then $cost(s) = \sum_j w(|i_j - i'_j|)$. The *interchange distance problem in w cost model* (or the w-interchange distance problem) is to compute $d_{WI(w)}(x, y) = \min\{cost(s) | s \text{ converts } x \text{ to } y\}$. If x cannot be converted to y we define $d_{WI(w)}(x, y) = \infty$. The *interchange distance problem* is simply the interchange distance problem in unit cost model.

Length-weighted cost models are recently claimed to be biological meaningful in genome rearrangements (see [13]). [13] studied the reversal rearrangement operator on permutation strings in ℓ^α cost models for different values of α. They showed that the sorting by reversals problem, which is known to be $\mathcal{NP}$-hard even on permutations, is polynomial time computable for some length-weighted cost models. Our results together with [13] might indicate a general phenomenon about length-weighted distances that should be further studied. The proposed classification of cost functions by the *laws of increasing/decreasing marginal cost* gives insight to the behavior of the different cost functions and enabled to derive results for cost functions that were not yet understood enough by [13]. To the best of our knowledge, this is the first paper that uses such a classification. It is the authors belief that these results are also guidelines for a better study of rearrangements cost models.

The paper is organized as follows. In Sect. 2 the unit cost model is studied, in which the distance is the minimum number of interchange operations needed

[3] The I-type and D-type classification of cost functions is in accordance with the traditionally called convex and concave functions. However, it is the authors opinion that the definition of the marginal cost laws gives better intuition to the results.

Table 1. A summary of results for the ℓ^α-interchange distance problem

α Value	Binary Alphabet	Permutations	General Strings
$\alpha = 0$	$O(m)$	$O(m)$	$\mathcal{NP}$-hard $O(m)$ 1.5-approximation
$0 < \alpha \leq \frac{1}{\log m}$	$O(m)$ *	$O(m)$ 2-approximation	$O(m)$ 3-approximation
$\frac{1}{\log m} < \alpha < 1$	$O(m)$ *	$O(m)$ 2-approximation	$O(m^3)$ $\lvert\Sigma\rvert$-approximation
$\alpha = 1$	$O(m)$	$O(m)$	$O(m)$
$1 < \alpha \leq \log 3$	$O(m)$	$O(m)$ 2-approximation	$O(m)$ 2-approximation
$\alpha > \log 3$	$O(m)$	$O(m)$	$O(m)$

$\star$ Only for the sorting problem. For the general rearrangement problem it is $O(m^3)$.

to rearrange one string into the other. This is the ℓ^α cost function, where $\alpha = 0$. For this cost model we show that the general problem is $\mathcal{NP}$-hard but fixed parameter tractable, and remains $\mathcal{NP}$-hard even if each symbol appears at most three times. We conclude with a 1.5-approximation algorithm. In Sect. 3 the ℓ_1-cost model is studied. A characterization of the distance is given, and it is proven to be polynomial time computable. In Sect. 4 and Sect. 5 the problem for $\mathcal{I}$-type and $\mathcal{D}$-type cost functions is studied, and optimal and approximation algorithms for the problem under these cost models are given. The results apply specifically to ℓ^α-interchange distance problem for every $\alpha > 1$ and $0 < \alpha < 1$, but apply to other functions as well (e.g. $\log(\ell)$). Omitted proofs will be given in the full paper.

2 Unit Cost Model

We begin by studying the problem in the unit cost model. We first show that this problem is equivalent to the problem of finding the cardinality of the maximum edge-disjoint cycle decomposition of Eulerian directed graphs (denoted by maxDCD). Then, we prove that the latter problem is $\mathcal{NP}$-hard. Given two strings, it is possible to derive two permutations of $1, \ldots, m$ by giving different labels to repeating symbols in both strings. The distance between the two permutations can then be viewed as the distance of a permutation of $1, \ldots, m$ from the identity permutation, for which we already know Fact 1. Of course, a different labelling yields different permutations of $1, \ldots, m$. Observation 1 specifies the connection between the interchange distance of the permutations and the interchange distance of the original strings.

Fact 1. [10] *The interchange distance of an m-length permutation π is $m - c(\pi)$, where $c(\pi)$ is the number of permutation cycles in π.*

Observation 1. *There exists a labelling for which the interchange distance between the resulting permutations of $1, \ldots, m$ is exactly the interchange distance between the given m-length strings. Moreover, the resulting permutations from this labelling have the minimum interchange distance over every other permutations resulting from any other labelling.*

Definition 2. *maxDCD is the following problem: Given an Eulerian directed graph $G = (V, E)$, find maximum-cardinality edge-disjoint directed cycle decomposition of G, i.e. partition of E into the maximum number of directed cycles.*

Lemma 1. *The interchange distance problem and the maxDCD problem can be linearly transformed to each other.*

Proof. We describe a linear transformation from the interchange distance problem to maxDCD. Let π_1, π_2 be two m-length strings over alphabet Σ, such that $|\Sigma| \leq m$, and π_1 is a permutation of π_2. We build the directed graph $G = (V, E)$, where $V = \Sigma$ and $E = \{e_i = (a, b) | 1 \leq i \leq m, \pi_1[i] = b, \pi_2[i] = a\}$. G is an Eulerian directed graph, since π_1 is a permutation of π_2, thus for every vertex in G the in- and out-degree are equal. The maximum edge-disjoint cycle decomposition of G includes only cycles with distinct vertices, since, if a vertex appears twice in a cycle, break it into two different cycles: one for each appearance of the repeating vertex (G is Eulerian). Thus, the proof is limited to cycles with distinct vertices. Every edge-disjoint cycle decomposition of G into cycles with distinct vertices is equivalent to a labelling of the strings symbols, resulting in two permutations of $1, \ldots, m$. For these permutations the graph cycles represent the permutations cycles. Denote the number of permutation cycles in a decomposition D_G by $c(D_G)$. By fact 1 their interchange distance is exactly $m - c(D_G)$. Finally, denote by $|maxDCD(G)|$ the cardinality of the maximum cycle decomposition of the directed graph G. Then, by Observation 1 the interchange distance between π_1 and π_2 is exactly $m - |maxDCD(G)|$.

The inverse transformation from maxDCD to the interchange distance problem for general strings is similar. Given an Eulerian directed graph $G = (V, E)$, we build π_1, π_2, two m-length general strings of symbols from alphabet Σ, where $m = |E|$ and $|\Sigma| = |V|$, such that π_1 is a permutation of π_2. Let $e_1, e_2, \ldots, e_{|E|}$ any order of the edges in G. For all $1 \leq i \leq m$, define $\pi_1[i] = b, \pi_2[i] = a$ if $e_i = (a, b)$. Since this is essentially the same transformation, only inversely built, we get as above that the interchange distance between π_1 and π_2 is exactly $m - |maxDCD(G)|$. □

Lemma 2. *The problem of finding a decomposition into directed triangles is polynomially reducible to maxDCD.*

Thus, it is enough to show is that directed triangle decomposition is $\mathcal{NP}$-hard. Following the $\mathcal{NP}$-completeness proof of edge-partition of undirected graphs into cliques of size k [14], we reduce the known $\mathcal{NP}$-complete 3SAT problem to our problem. The proof is similar to Holyer's [14] only we add the directions to the graph construction.

Theorem 1. *The directed triangles partition problem is $\mathcal{NP}$-hard.*

Corollary 1. *The maxDCD problem is $\mathcal{NP}$-hard.*

Theorem 2. *The interchange distance problem is $\mathcal{NP}$-hard.*

The graph construction used for the $\mathcal{NP}$-hardness proof gives a graph with in- and out-degree 3. By the proof of Lemma 1, in the instance of interchange distance problem transformed to this graph, symbols repeat 3 times. Corollary 2 follows. The hardness proof also implies Theorem 3. Lemma 3 is used to derive Theorem 4.

Corollary 2. *The interchange distance problem is $\mathcal{NP}$-hard even if every letter appears at most three times.*

Theorem 3. *The interchange distance problem is tractable for fixed $|\Sigma|$.*

Lemma 3. *Given an Eulerian directed graph $G = (V, E)$, then for every 2-cycle C in G there exist a partition of E into the maximum number of directed cycles, in which C appears as a cycle in the partition.*

Theorem 4. *There exists an $O(m)$ 1.5-approximation algorithm for the interchange distance problem.*

Proof. The proof of Lemma 1 implies that it is enough to find an approximation algorithm for the maxDCD problem. Given an Eulerian directed graph, we show how to find a cycle decomposition of C_{alg} cycles, and denote C_{opt} to be the maximal number of cycles. Note that, all the 2-cycles in a maximal cycle decomposition can be found in linear time, by inductively finding a 2-cycle and removing it, and applying Lemma 3 until there are no more 2-cycles in the graph. This suggest the following approximation algorithm. First find the 2-cycles as above, then arbitrarily decompose remaining edges into cycles.

We now prove the 1.5 approximation ratio of this algorithm. Denote by m the number of edges in the graph, and by C_2 the number of 2-cycles found by the algorithm. Clearly, $C_{alg} \geq C_2$. In an optimal cycle decomposition the remaining $m - 2C_2$ edges may form only cycles of length at least 3, and therefore, $C_{opt} \leq C_2 + \frac{m-2C_2}{3} = \frac{m+C_2}{3}$. By Fact 1, $d_{opt} \geq m - C_{opt} \geq \frac{2m-C_2}{3}$ and $d_{alg} \leq m - C_2$. Finally we get, $\frac{d_{alg}}{d_{opt}} \leq \frac{m-C_2}{\frac{2m-C_2}{3}} = \frac{m-C_2}{\frac{2}{3}(m-C_2)+\frac{C_2}{3}} \leq 1.5$. □

3 The ℓ_1-Cost Model

In this section we describe a characterization of the ℓ_1-interchange distance (denoted by WI(ℓ_1)) and a polynomial algorithm for the interchange distance problem in ℓ_1-cost model.

Permutation Strings. Let x, y be two strings with the same m distinct letters. Since the rearrangement of x to y can be viewed as sorting x by assuming y is the permutation $1, \ldots, m$, while making the appropriate changes for symbols names in x, in the sequel we assume that we sort x to the identity permutation. Lemma 4 is a first step in the characterization of the ℓ_1-interchange distance for permutations. We will show the connection to the ℓ_1-distance, defined below.

Definition 3. *Let x and y be general strings of length m. Let $\pi : [1..m] \rightarrow [1..m]$ such that for every i, $x[i] = y[\pi(i)]$, and define $cost(\pi) = \sum_{j=1}^{m} |j - \pi(j)|$. The ℓ_1-distance of x and y is: $d_{\ell_1}(x, y) = \min_\pi cost(\pi)$.*

Lemma 4. *Let π be a permutation of $1,\ldots,m$, and let I denote the identity permutation. Then, $d_{WI(\ell_1)}(\pi, I) \geq \frac{d_{\ell_1}(\pi, I)}{2}$.*

Proof. Denote by π_i the position of element i in π. In order to be sorted every element i must pass the *critical* segment $[\pi_i, i]$ if $\pi_i < i$ or $[i, \pi_i]$ if $i < \pi_i$. Note that in ℓ_1-cost model the distance is not diminished if element i passes the critical segment using more than one interchange in this segment. Thus, for every element we must pay its critical segment in the ℓ_1-cost model. Therefore, the best situation is where each interchange we perform sorts the two participating elements, since in this case we pay for each interchange exactly the cost that every element must pay. In this case, each interchange costs half the cost payed in the ℓ_1 distance, since in the ℓ_1 distance each element is charged independently. The lemma then follows. □

Given a permutation π of $1,\ldots,m$, we give a polynomial time algorithm that sorts the permutation by interchanges (see Fig. 1). Our algorithm performs only interchanges that advance both elements towards their final positions. The following definition is a formalization of this requirement.

Definition 4. *Let π be a permutation of $1,\ldots,m$. Let π_i denote the position of element i in π. A pair of elements i,j is a* good pair *if $j \leq \pi_i < \pi_j \leq i$.*

Lemma 5. *Every unsorted permutation has a good pair.*

Proof. By induction on k, the length of the permutation. For $k = 2$ the only unsorted permutation is $2, 1$, where the elements 1 and 2 are a good pair. Let π be a permutation of length k. We consider two cases.

Case 1: $\pi_k = k$. We can ignore the element k. The rest of the permutation is an unsorted permutation of length $k-1$, which by induction hypothesis has a good pair.

Case 2: $\pi_k < k$. We can delete the element k. If the resulting permutation is sorted then the pair k and π_k is a good pair. Otherwise, the resulting permutation of length $k-1$ has a good pair by induction hypothesis. It is easy to see that this pair is also a good pair in the original permutation of length k. □

Since the sum of costs of good-pairs interchanges never exceeds $d_{\ell_1}(\pi, I)/2$, Lemma 6 follows.

Lemma 6. *Let π be a permutation of $1,\ldots,m$. Then, $d_{WI(\ell_1)}(\pi, I) = \frac{d_{\ell_1}(\pi, I)}{2}$.*

Remark. Algorithm Sort requires $O(m^2)$ time. However, for computing the WI(ℓ_1)-distance (not an actual rearrangement sequence) an $O(m)$ algorithm is enough.

```
SORT(π)
Begin
    While there are unsorted pairs in π
        Find a good pair i,j.
        Interchange elements i and j.
End
```

Fig. 1. Algorithm Sort

General Strings. The main difficulty in the case of general strings is that repeating symbols have multiple choices for their desired destination. Let x and y be strings of length m. Our goal is to pair the locations in x to destination locations in y, so that repeating symbols can be labelled in x and y to get strings with the same m distinct letters (permutation strings). Such a labelling can be viewed as a permutation of the indices of x. Fortunately, we can characterize a labelling permutation of indices that gives the minimum ℓ_1-distance. This will be enough to derive a polynomial time algorithm for the ℓ_1-interchange distance problem in the general strings case as well.

Lemma 7. [Amir et al.][12] *Let $x, y \in \Sigma^m$ be two strings such that $d_{\ell_1}(x, y) < \infty$. Let π_o be the permutation that for any a and k, moves the k-th a in x to the location of the k-th a in y. Then, $d_{\ell_1}(x, y) = cost(\pi_o)$.*

Theorem 5. *Let x and y be m-length strings. Then, $d_{WI(\ell_1)}(x, y) = \frac{d_{\ell_1}(x,y)}{2}$.*

4 I-Type Cost Models

In this section we study the interchange distance problem in w-cost model for any $w \in \mathcal{I}$-type (denoted by WI(w)). The results apply for every ℓ^α-cost functions, $\alpha > 1$, as special case of $\mathcal{I}$-type. For $\alpha > \log 3$ we show an $O(m)$ optimal algorithm.

Permutation Strings. We deal with permutations (strings with distinct elements) first, and then show how to deal with general strings.

Theorem 6. *There exists an $O(m)$ 2-approximation algorithm for computing the WI(w)-distance on permutations, where $w \in \mathcal{I}$-type.*

Proof. An interchange on the segment $[i, j]$ can be mimicked by $2(j - i) - 1$ interchanges of size 1, as follows. First move $\pi[i]$ (the element at position i in π) to position j by $j - i$ interchanges of size 1, each advances $\pi[i]$ in one position. Now, the original $\pi[j]$ is one position closer to its final position, so $j - i - 1$ interchanges of size 1 bring $\pi[j]$ to its final position. All elements, except for the original $\pi[i]$ and $\pi[j]$, are in their original locations. We use the Sort algorithm (see Fig. 1), however, a good pair is sorted by $2(j - i) - 1$ interchanges of

size 1, as described above. Thus, the total cost of this algorithm is $cost_{alg} = \frac{\sum_{i=1}^{m} 2(|i-\pi_i|-1)w(1)}{2} \leq \sum_{i=1}^{m} |i-\pi_i| \cdot w(1)$. We now prove a lower bound on the cost of the optimal algorithm. Every element must pass its *critical* segment $[\pi_i, i]$ if $\pi_i < i$ or $[i, \pi_i]$ if $i < \pi_i$. The law of increasing marginal cost implies for every $a, b \in \mathbb{N}$, that $w(a) + w(b) \leq w(a+b)$, thus, to pass the critical segment the cost of $|i - \pi_i|$ interchanges of size 1 must be payed. Hence, $cost_{opt} \geq \frac{\sum_{i=1}^{m} |i-\pi_i| \cdot w(1)}{2}$. Therefore, the approximation ratio is: $\frac{cost_{alg}}{cost_{opt}} \leq 2$. □

Remark. Computing an actual rearrangement sequence using algorithm Sort is done in $O(m^2)$. Since Bubble Sort algorithm performs the minimum number of interchanges of size 1 as will be explained, it can also be computed using Bubble Sort.

General Strings. Again, in the case of general strings repeating symbols have multiple choices for their desired destination. However, as in the ℓ_1-cost model, we can use π_o to transform the strings into permutations of $1, \ldots, m$, and use Theorem 6. Theorem 7 follows.

Theorem 7. *There exists an $O(m)$ 2-approximation algorithm for computing the WI(w)-distance on general strings, where $w \in \mathcal{I}$-type.*

The Case $\alpha > \log 3$. We now show an optimal linear algorithm for $\alpha > \log 3$. The key observation here is that the optimal algorithm in this case never performs interchanges of size greater than 1, since every interchange on a segment $[i, j]$ where $(j-i) \geq 2$ can be mimicked by interchanges of size 1 in cost $2(j-i)-1$, which is less than $(j-i)^\alpha$. Compare the derivatives of $f(x) = 2x + 1$, which is 2, to the derivative of $h(x) = x^\alpha$, which is $\alpha x^{\alpha-1}$. For $\alpha > \log 3$ and $x > 2$ it holds $h'(x) > f'(x)$. Therefore, an optimal algorithm performs the minimum number of interchanges of size 1. [11] shows that this distance is characterized using the notion of *inversion* of a permutation described by Knuth [15]. Let π be a permutation of $1, \ldots, m$, define $I(\pi)$, the *inversion number* of π, by $I(\pi) = |\{(i, j) : 1 \leq i < j \leq m, \pi(i) > \pi(j)\}$. It is known that *Bubble Sort* requires precisely $I(\pi)$ interchanges of size 1 to sort any permutation π. Using same method as in Theorem 7 we get:

Theorem 8. *There exists an $O(m)$ algorithm for the w-interchange distance problem on general strings, where $w \in \ell^\alpha$, $\alpha > \log 3$.*

Binary Strings. For binary strings, when an interchange on the segment $[i, j]$ is mimicked by interchanges of size 1 in the proof of Theorem 6, only $j - i$ interchanges of size 1 are needed (and not $2(j-i)-1$) (because all the elements between i and j are 0's, so after we move $\pi[i]$ to position j by $j-i$ interchanges of size 1 there is nothing to fix). Thus the upper bound on the algorithm cost is equal to the lower bound on the optimal algorithm cost. We get:

Theorem 9. *There exists an $O(m)$ algorithm for the w-interchange distance problem on binary strings, where $w \in \mathcal{I}$-type.*

5 D-Type Cost Models

Binary Strings. Consider a limited version of the problem, the sorting problem, where all 0's should appear before the 1's. In binary string we need only pair the 1's that are in positions of 0's to the 0's that are in positions of 1's. For sorting binary strings Lemma 8 characterized a pairing that gives a minimal cost for every $w \in \mathcal{D}$-type. This gives Theorem 10.

Lemma 8. *Let x be an m-length binary string with b 0's. Let k be the number of unsorted 0's in x. Let π_r be the permutation that moves the k-th unsorted 0 in x to the location of the first unsorted 1 in x, the $k-1$ unsorted 0 in x to the location of the second unsorted 1 in x, ..., the first unsorted 0 in x to the location of the k-th unsorted 1 in x (i.e., π_r reverses the unsorted 0's and 1's). Then, for every $w \in \mathcal{D}$-type, $d_{WI(w)}(x, 0^b 1^{m-b}) = cost(\pi_r)$, where $cost(\pi) = \sum_{j=1}^{m} w(|j - \pi(j)|)$.*

Proof. Let π be a minimum cost pairing permutation. Assume to the contrary that $\pi \neq \pi_r$, and consider the left most 1 position j that is not paired as in π_r. Then, $j < \pi^{-1}(\pi_r(j)) < \pi(j) < \pi_r(j)$. We transform π into π' where j is paired to $\pi_r(j)$ and $\pi^{-1}(\pi_r(j))$ is paired to $\pi(j)$ (every other pairing unchanged). It holds: $cost(\pi) - cost(\pi') = w(|\pi(j) - j|) + w(|\pi_r(j) - \pi^{-1}(\pi_r(j))|) - w(|\pi_r(j) - j|) - w(|\pi(j) - \pi^{-1}(\pi_r(j))|) = w(|\pi(j) - j|) + w(|\pi_r(j) - \pi^{-1}(\pi_r(j))|) - w(|\pi_r(j) - \pi(j) + \pi(j) - j|) - w(|\pi(j) - \pi^{-1}(\pi_r(j))|) > 0$. Since by the law of decreasing marginal cost for every $w \in \mathcal{D}$-type $w(a + b + c) + w(b) < w(a + b) + w(b + c)$, for every a, b and c. Contradiction. □

Theorem 10. *There exists an $O(m)$ algorithm for the w-interchange sorting distance problem on binary strings, where $w \in \mathcal{D}$-type.*

Corollary 3. *There exists an $O(m)$ algorithm for the ℓ^α-interchange sorting distance problem on binary strings, where $0 < \alpha < 1$.*

Proof. We need only show that ℓ^α-cost functions for $0 < \alpha < 1$ have the decreasing marginal cost property, and get the result from Theorem 10. Note that for every $0 < \alpha < 1$ it holds that: $(a + b + c)^\alpha + b^\alpha < (a + b)^\alpha + (b + c)^\alpha$, for every a, b and c, because the derivative of $f(x) = x^\alpha$, $f'(x) = \alpha \cdot x^{\alpha-1}$ decreases when x increases. □

We now describe an algorithm for the general w-interchange distance problem on binary strings (but we only use it for $w \in \mathcal{D}$-type). Since in the binary case we need only pair the misplaced 0's to the misplaced 1's in minimum cost, we can find a minimum perfect matching between the misplaced 0's to misplaced 1's, represented as points in different sides of a bipartite graph, where an edge between a 0 and a 1 have the cost of the interchange between them according to w. For this problem use Edmonds-Karp algorithm [16] with the fibonacci heaps improvement of [17] to get an $O(m^3)$ algorithm. We conclude:

Theorem 11. *There exists an $O(m^3)$ algorithm for the w-interchange distance problem on binary strings, for every cost function w.*

Permutation Strings. We describe an $O(m)$ 2-approximation algorithm for permutation strings when $0 < \alpha < 1$. We first show a lower bound on the cost of the optimal algorithm. Every element must pass its critical segment in order to be sorted. Since $0 < \alpha < 1$ the distance is not diminished if element i passes the critical segment using more than one interchange in this segment. Thus, every element must pay at least its critical segment cost. Therefore, the cost where each interchange performed sorts the two participating elements must be payed. In this case, each interchange costs half the cost payed in the ℓ^α distance, $0 < \alpha < 1$, since in the ℓ^α distance each element is charged independently. Lemma 9 then follows.

Lemma 9. *Let π be a permutation of $1, \ldots, m$, and let I denote the identity permutation. Then, for every $0 < \alpha < 1$, $d_{WI(\ell^\alpha)}(\pi, I) \geq \frac{d_{\ell^\alpha}(\pi, I)}{2}$.*

Consider the Sort algorithm (see Fig. 1), restricted to choose only good pairs that sort at least one element. It is easy to show that every unsorted permutation has such a good pair by induction on the length of the permutation. By charging the sorted element on the interchange, the unsorted elements are charged only when they reach their final position. Since the algorithm only interchanges good pairs, elements are charged on a segment not greater than their critical segment, therefore, their contribution to the total cost of the algorithm is not greater than their contribution to the numerator of the lower bound stated in Lemma 9. Thus, we get Theorem 12.

Theorem 12. *There exists an $O(m)$ 2-approximation algorithm for the ℓ^α-interchange distance problem on permutation strings for every $0 < \alpha < 1$.*

General Strings. Let $\sigma_1, \ldots, \sigma_{|\Sigma|}$ be the alphabet symbols. First, use the matching algorithm of Edmonds-Karp to get a minimum cost sequence of interchanges that sorts all appearances of σ_1. An optimal algorithm pays at least this cost for sorting σ_1. After all appearances of σ_1 are sorted repeat the process for σ_2. There may be appearances of σ_2 that sorting σ_1 increased their cost compared to an optimal algorithm, however, the total increase cannot be greater than the total cost of the sorting of σ_1. Repeating the process for all alphabet symbols, when matching and sorting σ_i the total increase in the cost of the algorithm cannot be greater than the total cost of the sorting of $\sigma_1, \ldots, \sigma_{i-1}$. Thus, $cost_{alg} \leq cost_{opt}(\sigma_1) + [cost_{opt}(\sigma_1) + cost_{opt}(\sigma_2)] + [cost_{opt}(\sigma_1) + cost_{opt}(\sigma_2) + cost_{opt}(\sigma_3)] + \ldots + [cost_{opt}(\sigma_1) + \ldots + cost_{opt}(\sigma_{|\Sigma|})] \leq |\Sigma| cost_{opt}$. Also, note that $\sum_{i=1}^{|\Sigma|} \sharp(\sigma_i)^3 \leq \sum_{i=1}^{|\Sigma|} \sharp(\sigma_i) \cdot \sum_{i=1}^{|\Sigma|} \sharp(\sigma_i) \cdot \sum_{i=1}^{|\Sigma|} \sharp(\sigma_i) \leq m^3$, where $\sharp(\sigma)$ denotes the the number of appearances of σ. This proves Theorem 13.

Theorem 13. *There exists an $O(m^3)$ $|\Sigma|$-approximation algorithm for the ℓ^α-interchange distance problem on general strings for every $0 < \alpha < 1$.*

When α is small enough, an $O(m)$ 3-approximation algorithm for general strings can be designed. Let π be a permutation of $1, \ldots, m$. Consider the $O(m)$ algorithm for finding the unit cost distance of π described in [12]. Consider the sequence of interchanges in positions $(i_1, i'_1), \ldots, (i_k, i'_k)$ it performs, then,

$d_{WI(\ell^\alpha)}(\pi, I) = \sum_{j\in\{1..k\}} |i_j - i'_j|^\alpha \leq \sum_{j\in\{1..k\}} m^{\frac{1}{\log m}} = 2 \cdot d_{WI(\ell_0)}(\pi, I)$. On the other hand, $d_{WI(\ell^\alpha)}(\pi, I) \geq d_{WI(\ell_0)}(\pi, I)$. For general strings use the 1.5-approximation algorithm from Sect. 2 instead to get Theorem 14.

Theorem 14. *There exists an $O(m)$ 3-approximation algorithm for the ℓ^α-interchange distance problem on general strings for $0 < \alpha \leq \frac{1}{\log m}$.*

References

1. Bergeron, A.: A very elementary presentation of the Hannenhalli-Pevzner theory. Discrete Applied Mathematics 146(2), 134–145 (2005)
2. Caprara, A.: Sorting permutations by reversals and Eulerian cycle decompositions. SIAM journal on discrete mathematics 12(1), 91–110 (1999)
3. Hannenhalli, S., Pevzner, P.: Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. Journal of the ACM 46, 1–27 (1999)
4. Bafna, V., Pevzner, P.: Sorting by transpositions. SIAM J. on Disc. Math. 11, 221–240 (1998)
5. Hannenhalli, S.: Polynomial algorithm for computing translocation distance between genomes. Discrete Applied Mathematics 71, 137–151 (1996)
6. Christie, D.A.: Sorting by block-interchanges. Information Processing Letters 60, 165–169 (1996)
7. Heath, L.S., Vergara, P.C.: Sorting by short swaps. J. of Comp. Biology 10(5), 775–789 (2003)
8. Christie, D.A.: Genome Rearrangement Problems. PhD thesis, University of Glasgow (1999)
9. Radcliff, A.J., Scott, A.D., Wilmer, E.L.: Reversals and transpositions over finite alphabets. SIAM journal on discrete mathematics 19, 224–244 (2005)
10. Cayley, A.: Note on the theory of permutations. Philosophical Magazine 34, 527–529 (1849)
11. Jerrum, M.R.: The complexity of finding minimum-length generator sequences. Theoretical Computer Science 36, 265–289 (1985)
12. Amir, A., Aumann, Y., Benson, G., Levy, A., Lipsky, O., Porat, E., Skiena, S., Vishne, U.: Pattern matching with address errors: Rearrangement distances. In: Proc. 17th SODA, pp. 1221–1229 (2006)
13. Bender, M.A., Ge, D., He, S., Hu, H., Pinter, R.Y., Skiena, S., Swidan, F.: Improved bounds on sorting with length-weighted reversals. In: Proc. 15th SODA, pp. 912–921 (2004)
14. Holyer, I.: The NP-completeness of some edge-partition problems. SIAM Journal of Computing 10(4), 713–717 (1981)
15. Knuth, D.E: The Art of Computer Programming. In: Sorting and Searching, Reading, Mass, vol. 3, Addison-Wesley, London, UK (1973)
16. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmoc efficiency for network flow problems. Journal of The ACM (JACM) 19(2), 248–264 (1972)
17. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of The ACM (JACM) 34(3), 596–615 (1987)

Finding Mobile Data: Efficiency vs. Location Inaccuracy*

Amotz Bar-Noy[1,2] and Joanna Klukowska[2]

[1] Computer Science Department, Brooklyn College, CUNY
[2] Computer Science Department, The Graduate Center, CUNY
amotz@sci.brooklyn.cuny.edu
jklukowska@gc.cuny.edu

Abstract. A token is hidden in one out of n boxes following some known probability distribution and then all the boxes are locked. The goal of a searcher is to find the token in at most $D \leq n$ rounds by opening as few boxes as possible, where in each round any set of boxes may be opened. We design and analyze strategies for a searcher who does not know the exact values of the probabilities associated with the locked boxes. Instead, the searcher might know only the complete order or a partial order of the probabilities, or ranges in which these probabilities fall. We show that with limited information the searcher can find the token without opening significantly more boxes compared to a searcher who has full knowledge. This problem is equivalent to finding mobile users (tokens) in cellular networks (boxes) and finding data (tokens) in sensor networks (boxes).

Keywords: Wireless networks, sensor networks, location management, mobile computing.

1 Introduction

Consider the following combinatorial game. A *token* is placed and hidden in one out of n *boxes* following some probability distribution. The boxes are then locked and the only information about the location of the token is the probability distribution. A *searcher* needs to find the token as fast as possible while opening a minimum number of boxes. The searcher is given D, $1 \leq D \leq n$, rounds to find the token where, in each round the searcher may open any set of locked boxes.

More formally, let $C_1, C_2, \ldots, C_n$ be the n boxes and let $\bar{p} = \langle p_1, p_2, \ldots, p_n \rangle$, $\sum_{i=1}^{n} p_i = 1$, be the vector of probabilities associated with the n boxes. We call this vector the *prediction vector* or the *probability vector*. Without loss of generality, assume that $p_1 \geq p_2 \geq \cdots \geq p_n$. Let D be the number of *rounds* by which the token must be found. D is the *delay constraint* for the searcher. In particular, if the token is not found after $D-1$ rounds, then the searcher must open all the remaining boxes in the last round. Clearly, in order to minimize the

* This research was made possible by NSF Grant number 6531300.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 111–122, 2007.

number of opened boxes, the searcher should open the boxes following a non-decreasing order of their probabilities. That is, for $i < j$, any box opened at round i should be associated with at least as large a probability as that associated with a box opened at round j. Therefore, a search strategy is equivalent to partitioning the boxes into D disjoint sets (rounds). This partitioning is defined by the vector $\bar{d} = \langle d_0, d_1, d_2, \ldots, d_D \rangle$ where by definition $d_0 = 0$ and $d_D = n$. The meaning of this vector is that if the token was not found during the first $i-1$ rounds, then in round i the searcher opens the boxes $C_{d_{i-1}+1}, \ldots, C_{d_i}$.

Let ALG denote a search strategy for constructing the vector $\bar{d}$ based on the vector $\bar{p}$. The cost of searching for the token in D rounds using ALG is defined as the expected number of boxes opened until the token is found. If the token is in box C_i and that box is opened in round $j+1$ (i.e., $d_j < i \leq d_{j+1}$), then a searcher that follows the strategy ALG would open in total d_{j+1} boxes. Thus:

$$Cost_{ALG} = \sum_{j=1}^{D} \left(d_j \times \sum_{i=d_{j-1}+1}^{d_j} p_i \right). \quad (1)$$

The cost for any searcher is largest when $D = 1$ because all the boxes have to be opened in the first and only round and therefore the cost is n. On the other hand, the cost for an optimal searcher is minimized if it has $D = n$ rounds. In this case, the boxes should be opened one per round following the non-increasing order of their probabilities yielding a $\sum_{i=1}^{n} p_i i$ cost. For other values of D, the cost for D rounds is smaller than the cost for D' rounds for $n \geq D > D' \geq 1$. A simple formula for the cost does not exist for $1 < D < n$ rounds. Nevertheless, dynamic programming solutions exist that find the optimal partitioning in polynomial time for any $1 \leq D \leq n$, [1], [2], [3], [4].

In order to achieve the optimal partitioning, one needs to have an accurate knowledge of the prediction vector. The scope of this paper is to explore how the cost is affected when only partial information about the values of the probabilities is given. One motivation for exploring the tradeoff between such *location inaccuracies* and *search efficiency* is the fact that there exist situations in which there is no benefit in having the complete knowledge of the prediction vector. For example, if the number of allowed rounds is equal to the number of boxes ($D = n$), then a searcher who knows only the relative order of the probabilities is as efficient as a searcher who has complete knowledge. We show that there are other situations in which a searcher with a partial knowledge obtains results very close to a searcher who has a complete knowledge of probabilities.

Applications: This general search game was introduced for the problem of searching for a mobile user in a cellular network. In this setting, the token is a cellphone holder (user) and the boxes are the cells in which a particular user can be located. Opening a box is interpreted as paging the user in a cell using the wireless communication links. Finding the token fast is equivalent to finding the user while paging as few as possible cells.

A cellular phone system is composed of many cells and mobile users. The system has to be able to locate a mobile when phone calls arrive. The cost of delivering the call to a user is lower if the exact location of that user is known.

To that end, users can report their position whenever they cross boundaries of cells. However, reporting consumes wireless bandwidth (*uplink*) and battery power, and it is desirable for a user to report infrequently in order to save these resources. On the other hand, if the user does not report often enough, then when calls arrive, the system needs to perform a search that pages the user in each cell in which the user may be located. This method consumes the wireless bandwidth (*downlink*) resource. A compromise solution is a division of the cells into *location areas* (*zones*), each composed of several cells [5], [6]. Users report only when crossing boundaries of location areas. When a call to the user occurs, the system needs to search only within a given zone.

The cost of searching for a user depends upon the particular search strategy used. When a call for a user arrives, the system can either perform a *blanket search* in which all cells are paged at the same time, or a *sequential search* in which the cells are paged one after another. The first strategy guarantees that the user is found after one *round*, but uses the most downlink bandwidth. The second increases the number of rounds to possibly as many as the number of cells in the zone, but uses on average the minimum downlink bandwidth. Assuming that, at the time of the call arrival, the system knows the probabilities associated with every cell in the zone indicating the likelihood of the user being in that cell, the system can use a dynamic programming solution to find the user optimally for any given number of rounds.

The scope of this paper is to design efficient search strategies for locating a user even when the prediction vector is not accurate. The inaccuracy can be caused by, for example, limited tracking methods used by the system, large irregularities in users' mobility patterns, or even intentional decreases in accuracy of location information in an attempt to give the user more *privacy*. Sometimes an accurate prediction vector reveals a lot of information about the user and may even open up potential security risks. Some users may prefer to maintain a higher level of privacy even at a higher price. For an individual user, the delay in delivery of a call that occurs when the system needs to page all possible locations is not noticeable and not important as long as the call is received. On the other hand, keeping one's whereabouts private may be valued a lot. We show that the system can offer more privacy to its users by storing only an approximation to the location probability vector without paying significantly more for delivering each call.

Another application is in wireless sensor networks. The sensors are not mobile, however the system does not always know where to find specific data that is gathered by the sensors. Hence, one can view this data as mobile data. Nevertheless, the system might know the probability (or an estimate probability) of finding the answer at each sensor. One can view the sensors as the boxes and the answer to the query as the token. Opening few boxes is translated into communicating with few sensors which is a paramount objective in sensor networks due to battery limitations for the sensors. Using only a few rounds is equivalent to finding an answer to a query quickly.

Our Contributions: We analyze and evaluate the performance of strategies for a searcher that has limited information about the probability vector. We compute the *worst case ratio*, ρ, between algorithms that are based on some limited information and the optimal solution that relies on complete knowledge of the prediction vector. We also compare the solutions via a simulation assuming the Zipf distribution for the prediction vector. Let $Opt(D)$ be the cost of the algorithm that has complete knowledge of the probability vector and has D rounds to find the token.

Our first contribution is the analysis of the case where the searcher knows only the relative order of the values in the prediction vector. Following [7], we propose algorithm *DRoot* whose performance in the worst case is $\Theta(\sqrt[D]{n})Opt(n)$ and we prove that no other algorithm can perform better. According to the performance analysis results, the cost computed by this algorithm is much closer to the optimal solution. The performance of the algorithm depends on the allowed delay in finding the token. When that delay is equal to the number of boxes, *DRoot* finds a solution with optimal cost.

Our second contribution is the analysis of various scenarios with only partial knowledge of the prediction vector. In general the probabilities are assigned to B bins, where the relative order of the probabilities between the bins is known but within a bin the relative order of the probabilities is unknown. We analyze the performance of only sequential search given that limited information. We discuss several rules for bin assignment. *Assignments based on size* considerations guarantee a cost that is $\frac{n}{B}\,Opt(n)$ in the case of bins with the same size, and $\Theta(n^{1/B})\,Opt(n)$ when the sizes of bins vary exponentially. Using *assignments based on threshold* considerations, the computed worst case costs are similar: $\Theta(\frac{n}{B})\,Opt(n)$ for *linear threshold bins*, and $\Theta(n^{1/B})\,Opt(n)$ for *exponential threshold bins*. For *assignments based on optimal solution for 2 rounds*, the worst case cost for a searcher is $O(\sqrt{n})\,Opt(n)$. Our simulation results demonstrate that the theoretical bounds are truly limited to the worst case and on average the cost computed with the limited knowledge offered by different types of bins is close to optimal.

Related Work: Modeling uncertainty of location of mobiles as a probability distribution vector was studied in [8], which also presented a framework for measuring uncertainty. Location management schemes were discussed in [5], [6]. The optimal solution for the partitioning of n cells into $1 \leq D \leq n$ rounds was proposed by [1], [2], [3]. Recently the running time of the dynamic programming method has been improved by the authors of [4]. In [9] the authors presented a suboptimal solution which is computationally more efficient than dynamic programming. Other variations of the problem have been proposed. One of them is paging multiple users for a conference call, [10]. Another is the added effect of queuing requests to search for single users and performing a search for several mobiles at once, [2], [11], [12].

Privacy issues have been raised in recent years regarding the tradeoff between privacy and the benefits of location based applications [13], [14], [15]. Most of the research is based on assuming that the wireless carrier is a trusted party

and developing algorithms to limit access of third parties to the location data of mobiles, [14], [15]. In [13] the author provides a survey of the technology, issues, and regulations related to location-aware mobile devices.

2 Relative Order

One of the motivations for this research was the fact that knowing only the relative order of probabilities associated with the boxes is enough to enable the sequential search strategy (i.e., $D = n$) to achieve optimal cost. In this case, the searcher can find a token with the same cost as if he/she knew the exact value of the prediction vector. In this section, we decrease the allowed delay and examine how well the searcher can perform. Note that knowing only relative order makes $\langle 1, 0, \ldots, 0\rangle$ and $\langle 1/n, \ldots, 1/n\rangle$ indistinguishable. A searcher cannot achieve the optimal cost since the strategies for opening these boxes associated with the two vectors may be very different. The authors of [7] show that for $D \geq \log_2(n)$ the searcher can find the token with cost at most two times larger than an optimal solution that uses complete knowledge, and that for $D < \log_2(n)$ the cost is $\Theta(n^{1/D})Opt(n)$.

Let *DRoot* be the algorithm that creates D partitions by assigning $d_i = \lfloor n^{i/D} \rfloor$ for $1 \leq i \leq D$.

Theorem 1 (from [7]). *The cost of searching for a token, given only the relative ordering of the boxes, computed by* DRoot *using at most D rounds is $O(n^{1/D})\ Opt(n)$ in the worst case.*

In the next theorem, we show that *DRoot* is optimal.

Theorem 2. *The lower bound on the performance in the worst case for any algorithm that is given only the relative order of the probabilities associated with the boxes is $\Omega(n^{1/D}\, Opt(D))$, for $D < \log_2(n)$.*

Proof. Let $\langle d_1, d_2, \ldots, d_D\rangle$ be the partition vector used by an arbitrary algorithm, *ALG*. By the pigeon hole principal, there is at least one d_j such that $d_j/d_{j-1} \geq n^{1/D}$ (for $j = 1$, assume that $d_0 = 1$; this allows us to treat d_1 the same as other $d'_j s$). The assumption that $D < \log_2 n$ implies that $n^{1/D} > 2$ and therefore $d_j > 2d_{j-1}$. Consider a distribution that has $2d_{j-1}$ boxes each with probability $1/(2d_{j-1})$ and the remaining boxes with probability equal to zero. For this distribution the algorithm with the above partitions assigns d_{j-1} boxes with nonzero probability to the j^{th} round. Ignoring the cost that this algorithm has to pay for the previous $j-1$ rounds, its total cost is:

$$Cost_{ALG} \geq \left(\tfrac{1}{2d_{j-1}} d_{j-1}\right) d_j = \tfrac{1}{2} d_j = \tfrac{1}{2} d_{j-1} n^{1/D} \ .$$

The optimal cost has to be no greater than the solution for 2 rounds in which the first d_{j-1} boxes are opened in the first round, and the next d_{j-1} boxes are opened in the second round. Therefore,

$$Cost_{Opt} \leq \left(\tfrac{1}{2d_{j-1}} d_{j-1}\right) d_{j-1} + \left(\tfrac{1}{2d_{j-1}} d_{j-1}\right) 2d_{j-1} = \tfrac{3}{2} d_{j-1} \ .$$

These two costs give a lower bound on the ratio of any algorithm and the optimal solution: $\rho \geq n^{1/D}/3$.

Remark: For $D = 2$ it can be shown that the lower bound is larger, $\rho \geq n^{1/D}/\sqrt{3}$

3 Bins

A natural next step in limiting the knowledge available to a searcher is a *prediction bin* strategy. We represent the knowledge of a searcher by bins that are groupings of boxes based on some characteristic. The motivation for this limitation in the applications described in the introduction is that it might be easier to estimate the values of the probabilities this way, than trying to arrange them in the complete relative order. The B *bins*, $b_1, b_2, \ldots, b_B$, are a partition of the set of boxes, $C_1, C_2, \ldots, C_n$, with probabilities $p_1, p_2, \ldots, p_n$, such that $C_i \in b_k$ and $C_j \in b_{k+1}$ implies that $p_i \geq p_j$, where $1 \leq i, j \leq n$ and $1 \leq k < B$. We do not know anything about the relationship of boxes that are in the same bin; i.e., if $C_i, C_j \in b_k$, then $p_i \geq p_j$ or $p_i < p_j$.

We consider several criteria for putting boxes into bins. In *size bin*, each bin contains some predefined number of boxes, for example, uniform size with $\frac{n}{B}$ boxes per bin. When $B = n$ this uniform size bin assignment is equivalent to the relative order case discussed in the previous section. In *threshold bin*, the boundaries of the bins are assigned based on some threshold values. For example, with $B = 2$ and threshold $\frac{1}{n}$, all the boxes with probability greater then or equal to $\frac{1}{n}$ are placed into b_1 and all the boxes with probability less then $\frac{1}{n}$ are placed into b_2. Note that some bins can be empty. In *solution for D bin*, the $B = D$ bins are based on the optimal partition of n boxes into D rounds.

In this section we consider only sequential search using various bin assignments. Given the set of *prediction bins* there is essentially one way to search, since the boxes within a bin are indistinguishable from each other; the only rule that any smart solution should follow is to search the bins according to their order. In the analysis of the various types of bin strategies that follow, we call the procedure for opening boxes using *prediction bins*, ALG, and the cost obtained by it $Cost_{ALG}$.

From the application point of view, consider the wireless cellular network that provides some privacy to its users. The bins reveal much less information about a mobile's location than a full relative order among the cells does. With bins, the system knows the relative order only among the bins, but within each bin, it has no way of deciding what the order of cells is.

3.1 Size Bins

The idea of size bins is that the algorithm is given B bins with specific sizes. When searching within a single bin, the worst case is when the searcher inadvertently opens the boxes within that bin in reverse order starting with the one with the smallest probability, going to the largest. Performance in the worst case

depends on how the sizes of particular bins are chosen. If this happens in the first bin (the one that contains the boxes with highest probabilities) and that bin has large size, then the boxes with high values contribute to the cost a lot more than they do if the opening process is done in the correct order. We provide matching bounds for the performance of a searcher in the worst case.

Linear Size Bins. In this strategy, the sizes of all bins differ by at most one. The first $n - (n \bmod B)$ bins have size $\lfloor n/B \rfloor$ and the remaining bins have size $\lceil n/B \rceil$. For simplicity of computation assume that n is a multiple of B; then $|b_1| = |b_2| = \ldots = |b_B| = \frac{n}{B}$. The bins with lower index numbers contain the boxes with higher probability, i.e., $C_i \in b_k$ and $C_j \in b_{k+1}$ implies that $p_i \geq p_j$ for any $1 \leq i, j \leq n$ and $1 \leq k < B$. For an arbitrary bin b_i, the boxes in it are opened in n/B rounds one after another without knowing which are opened first.

Theorem 3. *Any algorithm that opens boxes in n rounds given linear size prediction bins achieves, in the worst case, a cost of* $\frac{n}{B}\,Opt(n)$.

Proof Sketch. The worst case occurs when ALG opens the boxes placed in a single bin from the one with smallest probability to largest. Then $Cost_{ALG} \leq \frac{n}{B}\pi_1 + \frac{2n}{B}\pi_2 + \ldots + \frac{Bn}{B}\pi_B$, where π_i is the sum of all the probabilities in a bin b_i. The optimal algorithm that has complete knowledge of the prediction vector opens the boxes from each bin in decreasing order, so its cost is $Cost_{Opt} \geq 1\pi_1 + \frac{n}{B}\pi_2 + \ldots + \frac{(B-1)n}{B}\pi_B$. This guarantees a ratio $\rho \leq n/B$.

The worst case is met when the prediction vector is $\langle 1, 0, \ldots, 0 \rangle$. Optimally the box with probability 1 should be open first, but a searcher may open it as the last box in the first bin causing the worst case ratio $\rho = n/B$.

Exponential Size Bins. In the exponential bin strategy, the size of a bin b_i is $n^{i/B} - n^{(i-1)/B}$, where $1 \leq i \leq B$. For simplicity assume that $n = k^B$ for some positive integer k. This type of bin assignment puts the boxes with higher probability into smaller bins, thereby reducing the potential cost of the worst case in which the boxes in the first few bins are opened in reverse order.

The proof of the next theorem is in the same style as the one for Theorem 3.

Theorem 4. *Any algorithm that opens the boxes in n rounds given exponential size prediction bins achieves, in the worst case, a cost of* $\Theta(n^{1/B})\,Opt(n)$.

3.2 Threshold Bins

In the threshold bin strategy, bins are assigned probability ranges, and boxes with probabilities within a bin's range are assigned to that bin. Consequently, it is possible to have empty bins. Like the size bin strategy, within any bin one does not know the relative values of the probabilities. Ideally one would assign bin boundaries in such a way that the boxes are nearly evenly distributed among the bins, but there is no way to do this. A bad choice might place all the boxes into a single bin. In practice, this method leaves a lot of bins empty, which leads to bad performance. As for size bins, we provide matching bounds for the performance of a searcher in the worst case.

Linear Threshold Bins. The simplest threshold bin strategy is the one in which the width of the range of every bin is $\frac{1}{B}$, i.e., the bins would be: $(1-\frac{1}{B},1]$, $(1-\frac{2}{B},1-\frac{1}{B}]$, ..., $(\frac{1}{B},\frac{2}{B}]$, $[0,\frac{1}{B}]$. Thus, bin b_i has boxes with probabilities p_i such that $\frac{B-i}{B} < p_i \leq \frac{B-i+1}{B}$. The worst case occurs when all of the boxes are placed into the last bin (the one with range $[0,\frac{1}{B}]$), because no information about relative values of probabilities is revealed.

Theorem 5. *Any algorithm that opens the boxes in n rounds given linear threshold prediction bins achieves, in the worst case, a cost of* $\Theta(\frac{n}{B})$ *Opt*(n).

Proof Sketch. Denote by z_i the number of boxes in the bins with indices smaller than i and by x_i the number of boxes in the bin b_i. No matter what the values of z_i and x_i are, the costs for bin i are as follows:

$$Cost_{ALG} \leq \tfrac{B-i+1}{B} \textstyle\sum_{j=z_i+1}^{z_i+x_i} j \qquad \textit{and} \qquad Cost_{Opt} \geq \tfrac{B-i}{B} \textstyle\sum_{j=z_i+1}^{z_i+x_i} j \ .$$

The worst case for the game with only linear threshold prediction bins is $\rho < \frac{B-i+1}{B-i} < 2$ for $1 \leq i \leq B-1$. The last bin requires special consideration. Let x_{B+} be the number of boxes in the last bin with nonzero probability and x_{B0} be the number of boxes in b_B that have a probability of 0. The above costs become:

$$Cost_{ALG} \leq \tfrac{1}{B} \textstyle\sum_{j=z_B+x_{B+}+x_{B0}+1}^{z_B+x_{B+}+x_{B0}} j \qquad \textit{and} \qquad Cost_{Opt} \geq p_{min} \textstyle\sum_{j=z_B+1}^{z_B+x_{B+}} j \ ,$$

in which p_{min} is the smallest nonzero probability associated with a box in b_B. Using the fact that $z_B + x_{B+} + x_{B0} = n$ the ratio can be simplified to $\rho \leq (2n - x_{B+} + 1)/(Bp_{min}(2z_B + x_{B+} + 1))$. This ratio has maximum value when $z_B = 0$, for which $\rho = O(n/B)$. The ratio of the costs of two algorithms is less then or equal to the maximum ratio of the costs of the two algorithms for any given bin, so it is less than or equal to the worst ratio for the last bin, $\rho = O(n/B)$.

This worst case is met by the prediction vector $\langle \frac{1}{B}-\epsilon', \ldots, \frac{1}{B}-\epsilon', \epsilon, \ldots, \epsilon \rangle$. Without affecting much the value of ρ, both ϵ' and ϵ may be replaced by zero. If a searcher opens boxes in order of nondecreasing probabilities, then the ratio of the cost of this searcher to that of an optimal algorithm is $\rho \geq \frac{2n-B+1}{B+1} = \Omega(\frac{n}{B})$.

Exponential Threshold Bins. The exponential threshold bin strategy takes advantage of the fact that there can be only a few boxes with large probabilities but possibly many with very small probabilities. The exponential threshold bin strategy provides finer granularity of bins for the boxes with smaller probabilities. This makes it less likely that all the boxes are placed into one or just a few bins. Bin b_i contains all the boxes with probabilities in the semi-closed range from $n^{-(i-1)/B}$ to $n^{-i/B}$, to be precise, the bins are: $[n^{-1/B}, 1]$, $[n^{-2/B}, n^{-1/B})$, ..., $[0, n^{-(B-1)/B})$.

The proof of the next theorem is in the same style as the one for Theorem 5.

Theorem 6. *Any algorithm that opens the boxes in n rounds given exponential threshold prediction bins achieves, in the worst case, a cost of* $\Theta(n^{1/B})$ *Opt*(n).

3.3 Optimal Solution Bins

This bin assignment strategy assumes that somehow the searcher knows the values of the optimal partitions for D rounds and needs to find the token in $D' \neq D$ rounds. Specifically we analyze the case of going from a solution for 2 rounds to a sequential search, i.e., n rounds, and provide an upper bound on the worst case performance of a searcher. Other cases remain open problems and we present only simulation results for them in Section 4.

Solution for 2 Paging Rounds. Given the solution for 2 rounds, the only information present is about which boxes are opened in which round. It is known that the threshold between two partitions is between $1/2n$ and $2/n$, [7]; hence, the searcher can approximate the location of the threshold that separates two bins.

Theorem 7. *Any algorithm that opens the boxes in n rounds given prediction bins that follow the optimal partition for two rounds achieves, in the worst case, a cost of $O(\sqrt{n})$ $Opt(n)$.*

We use the above-mentioned characteristics of the optimal partitioning into two paging rounds. This allows us to analyze two cases depending on the value of the optimal partition. In the first, the worst case for an arbitrary algorithm is caused by opening boxes in the second bin in the wrong order, in the second case an algorithm has to pay more for opening in the wrong order the boxes from the first bin. In both cases we show that the ratio of that algorithm to the optimal cannot be more than $O(\sqrt{n})$. The detailed proof is beyond the space limitations of this publication.

4 Performance Analysis

We implemented all of the proposed algorithms and strategies and compared all results. We ran the algorithms on different combinations of the input parameters n, D and B for various distributions. We present here just a selection of the results. We mainly used the *Zipf* distribution, [16], which is observed in many commonly occurring phenomena. The values of the probabilities associated with the boxes according to this distribution are proportional to $(1/i)^\alpha$ for $1 \leq i \leq n$ and a parameter $\alpha \geq 0$ and normalized by $\sum(1/j)^\alpha$ so that $\sum_{j=1}^{n} p_j = 1$. When the parameter α is equal to zero the distribution is uniform (all the values are the same), and as α increases the values become skewed towards the p_i's with lower indices. When $\alpha = 3$, the value of p_1 is already very close to 1 and the remaining p_i's are negligible.

The first set of tests examined the performance of algorithm *DRoot* that knows only the relative order of probabilities of the token being located in each box. The numerical results show that the performance is very close to the optimal algorithm that knows the entire probability vector. The chart in Figure 1 shows that the cost computed by *DRoot* algorithm remains very close to the cost of the optimal algorithm as the number of boxes increases. We ran the algorithm

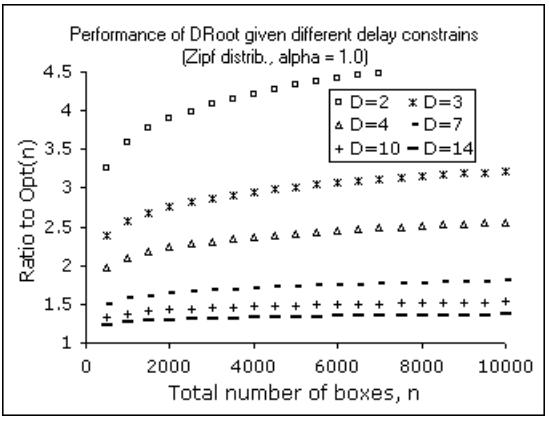
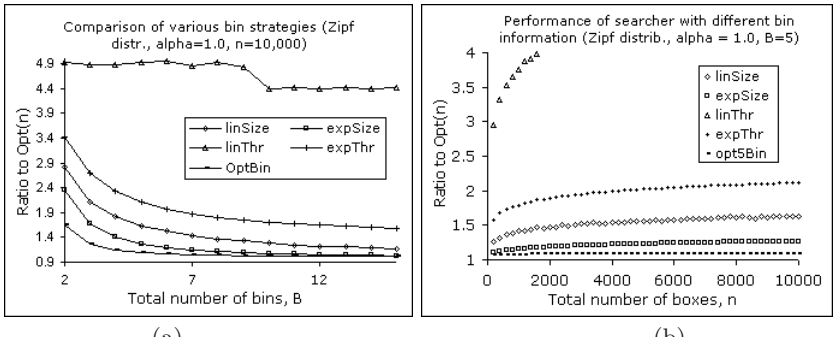


Fig. 1. The performance of *DRoot* algorithm improves as D increases

Table 1. The ratio of cost of the *DRoot* algorithm to the cost of an optimal solution for different values of the delay (D) and different values of parameter α to Zipf distribution in comparison to the theoretical worse case of that ratio

n =	Parameter to Zipf distribution, α											Theor.
10,000	0.4	0.6	0.8	1	1.2	1.4	1.6	1.8	2	2.2	2.4	worst case
D=2	1.438	1.500	1.541	1.531	1.449	1.298	1.123	1.010	1.033	1.221	1.569	100.000
D=4	1.524	1.543	1.537	1.496	1.411	1.282	1.133	1.026	1.037	1.219	1.577	10.000
D=6	1.435	1.434	1.417	1.378	1.312	1.218	1.109	1.023	1.019	1.144	1.405	4.642
D=8	1.357	1.350	1.331	1.298	1.247	1.174	1.090	1.025	1.033	1.171	1.409	3.162
D=10	1.299	1.290	1.273	1.245	1.203	1.144	1.075	1.020	1.023	1.111	1.255	2.512
D=12	1.256	1.247	1.232	1.208	1.172	1.123	1.067	1.025	1.043	1.134	1.277	2.154
D=14	1.223	1.215	1.201	1.180	1.149	1.107	1.057	1.013	1.001	1.029	1.071	1.931



(a) (b)

Fig. 2. The performance of the 5 bin strategies: linear (linSize) and exponential (expSize) size bins, linear (linThr) and exponential (expThr) threshold bins, and optimal solution bin (OptBin) for (a) increasing number of bins, (b) increasing number of boxes

on input sizes of up to $n = 10,000$ to see if the performance is affected by the large number of boxes. We found out that *DRoot* performs well independent of the input size. Table 1 shows more specific results for $n = 10,000$. Independent of the parameter α and the number of rounds allowed, D, algorithm *DRoot* performs much better than suggested by the theoretical analysis. Similar results were observed with other distributions (e.g. Gaussian).

The second set of tests compared the results of different bin strategies. We compared five different types of inaccurate prediction vectors. The tests showed that even with a very small number of bins the cost obtained by the searcher that is given only knowledge about the bins is very close to the optimal cost. The chart in Figure 2(a) shows these results for the Zipf distribution with parameter $\alpha = 1.0$ and $n = 10,000$. The results for smaller values of n were even closer to optimal. The tests on other distributions showed also that the searcher only needs a small numbers of bins to obtain good results. The chart in Figure 2(b) demonstrates the results obtained by the searcher with only 5 bins for different types of bins as the number of boxes increases. For all the Zipf distributions the *optimal solution for D bin* gave the cost closest to optimal.

5 Open Problems

We have presented several ways of finding a token in one out of n boxes, when the knowledge about the prediction vector associated with these boxes is limited. The limited information increases the cost of searching for the token, but the increase is not significant even in the worst case. The performance analysis shows that on average the increase in cost is much smaller than the one suggested by the worst case analysis. There are several questions that still remain open (some of which are work in progress).

In section 2, we showed a lower bound of the worst case behavior of any algorithm that knows only the relative order of probabilities. Narrowing the gap between the upper bound of algorithm *DRoot* and the general lower bound for $D \geq 3$ as well as matching the two bounds for $D = 2$ remain open problems.

In Section 3, we discussed the performance of sequential search given various types of bins. An interesting research topic is to provide similar analysis of cases in which the delay constraint is $D < n$. In Subsection 3.3, we analyzed the sequential search performance of a searcher who knows the optimal solution for $D = 2$ rounds. We encountered similar problems as the authors of [7] with the analysis for the case of $D \geq 3$ rounds. A matching lower bound for the case of $D = 2$ is also work in progress.

Other bin assignments can be introduced and analyzed, for example a combination of size type bins and threshold type bins, strategies with a variable number of bins, or bins whose content adds up to some predetermined constant.

In practice, our results may suggest a method of measuring the privacy level offered by the camouflaging strategies used by cellular networks. The question is whether the optimality of a strategy could be an indication of privacy that it provides to the user. That is, if the system can find the user with cost close

to optimal, does it follow that the strategy does not guarantee much privacy? This type of analysis requires a specific definition of privacy that includes some metrics by which it can be measured.

Acknowledgments. We thank the reviewers of the previous version of the paper for the ideas and comments that were incorporated into this version.

References

1. Madhavapeddy, S., Basu, K., Roberts, A.: Adaptive paging algorithms for cellular systems. Kluwer Academic Publishers, Norwell, MA, USA (1996)
2. Goodman, D.J., Krishnan, P., Sugla, B.: Minimizing queuing delays and number of messages in mobile phone location. Mobile Networks and Applications 1(1), 39–48 (1996)
3. Krishnamachari, B., Gau, R.-H., Wicker, S.B., Haas, Z.J.: Optimal sequential paging in cellular wireless networks. Wirel. Netw. 10(2), 121–131 (2004)
4. Bar-Noy, A., Feng, Y.: Efficiently paging mobile users under delay constraints. In: Proc. of 26th IEEE Conf. on Computer Communications, May 2007, pp. 1910–1918. IEEE Computer Society Press, Los Alamitos (2007)
5. Jain, R., Lin, Y.B., Mohan, S.: Location Straegies for Personal Communications Services. Mobile Communications Handbook, ch.18. CRC Press, Boca Raton, USA (1996)
6. Akyildiz, I., McNair, J., Ho, J., Uzunalioglu, H., Wang, W.: Mobility management in next-generation wireless systems. In: Proceedings of the IEEE, vol. 87, pp. 1347–1384. IEEE Computer Society Press, Los Alamitos (1999)
7. Bar-Noy, A., Mansour, Y.: Competitive on-line paging strategies for mobile users under delay constraints. In: Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, pp. 256–265. ACM Press, New York (2004)
8. Rose, C., Yates, R.: Location uncertainty in mobile networks: a theoretical framework. Communications Magazine, IEEE 35(2), 94–101 (1997)
9. Wang, W., Akyildiz, I.F., Stuber, G.: An optimal partition algorithm for minimization of paging costs. IEEE Comm. Letters 5(2), 42–45 (2001)
10. Bar-Noy, A., Malewicz, G.: Establishing wireless conference calls under delay constraints. J. Algorithms 51(2), 145–169 (2004)
11. Rose, C., Yates, R.: Ensemble polling strategies for increased paging capacity in mobile communication networks. Wirel. Netw. 3(2), 159–167 (1997)
12. Gau, R.-H., Haas, Z.J.: Concurrent search of mobile users in cellular networks. IEEE/ACM Trans. Netw. 12(1), 117–130 (2004)
13. Minch, R.P.: Privacy issues in location-aware mobile devices. In: Proceedings of the 37th Hawaii International Conference on System Sciences (2004)
14. Gruteser, M., Liu, X.: Protecting privacy in continuous location-tracking applications. IEEE Security and Privacy Magazine 02(2), 28–34 (2004)
15. Hoh, B., Gruteser, M.: Protecting location privacy through path confusion. In: Proceedings of the 1st International Conference on Security and Privacy for Emerging Areas in Communications Networks, pp. 194–205 (2005)
16. Zipf, G.K.: Human Behaviour and the Principle of Least Effort. Addison-Wesley, London, UK (1949)

A Faster Query Algorithm for the Text Fingerprinting Problem

Chi-Yuan Chan, Hung-I Yu, Wing-Kai Hon, and Biing-Feng Wang

Department of Computer Science,
National Tsing Hua University,
Hsinchu, Taiwan 30043,
Republic of China
{jous,herbert,wkhon,bfwang}@cs.nthu.edu.tw

Abstract. Let S be a string over a finite, ordered alphabet Σ. For any substring S' of S, the set of distinct characters contained in S' is called its *fingerprint*. The *text fingerprinting problem* consists of constructing a data structure for the string S in advance, so that on given any input set $C \subseteq \Sigma$ of characters, we can answer the following queries efficiently: (1) determine if C represents a fingerprint of some substrings in S; (2) find all maximal substrings of S whose fingerprint is equal to C. The best results known so far solved these two queries in $\Theta(|\Sigma|)$ and $\Theta(|\Sigma| + K)$ time, respectively, where K is the number of maximal substrings. In this paper, we propose a new data structure that improves the time complexities of the two queries to $O(|C| \log(|\Sigma|/|C|))$ and $O(|C| \log(|\Sigma|/|C|) + K)$ time, respectively, where the term $|C| \log(|\Sigma|/|C|)$ is always bounded by $\Theta(|\Sigma|)$. This result answers the open problem proposed by Amir *et al.* [A. Amir, A. Apostolico, G.M. Landau, G. Satta, Efficient text fingerprinting via Parikh mapping, J. Discrete Algorithms 1 (2003) 409–421]. In addition, our data structure uses less storage than the existing solutions.

Keywords: Fingerprints, Combinatorial algorithms on words, Text indexing, Patricia trie.

1 Introduction

Consider a string S over a finite, ordered alphabet Σ. Let $|S| = n$. For any substring S' of S, the set of distinct characters contained in S' is called the *fingerprint* of S', and we say such a fingerprint *appears* in S. The *text fingerprinting problem*, which was first introduced by Amir *et al.* [1], consists of the following two sub-problems:

Find_All: Compute all fingerprints that appear in S;
Indexing: Construct a data structure for S in advance, so that we can answer several queries about the fingerprints that appear in S efficiently.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 123–135, 2007.

For the indexing problem, the following queries are studied in the literature:

Query 1: Given a set $C \subseteq \Sigma$, answer if C is a fingerprint in S.
Query 2: Given a set $C \subseteq \Sigma$, find all maximal locations of C in S, where a maximal location of a fingerprint F corresponds to a maximal substring whose fingerprint is F.

Efficient solutions to these problems have important applications in the fields of natural language processing [1,5], computational biology [2,9] and formal languages [1]. Amir *et al.* first proposed an $O(n|\Sigma|\log|\Sigma|\log n)$-time algorithm for the Find_All problem based on the interesting *naming technique* [1]. Using this algorithm as a preprocessing step, they gave a data structure that answers Query 1 and Query 2 in $O(|\Sigma|\log n)$ time and $O(|\Sigma|\log n+K)$ time, respectively, where K is the number of maximal locations of C. The space of their data structure is $\Theta(n|\Sigma|\log|\Sigma|)$. Didier *et al* [3] improved the time for Find_All to $\Theta(n|\Sigma|\log|\Sigma|)$, while keeping the same query times and space for the indexing problem. Besides, they also proposed a $\Theta(n^2)$-time algorithm for Find_All.

Let $\mathcal{F}$ denote the set of all distinct fingerprints in S and let $\mathcal{L}$ denote the set of all maximal locations of all fingerprints in $\mathcal{F}$. Based on $\mathcal{F}$ and $\mathcal{L}$ as parameters, Kolpakov and Raffinot [7] proposed a $\Theta((n+|\mathcal{L}|)\log|\Sigma|)$-time algorithm for Find_All, and constructed an elegant data structure called the *fingerprint tree* to answer Query 1 and Query 2 in $\Theta(|\Sigma|)$ time and $\Theta(|\Sigma|+K)$ time, respectively. Their data structure can be constructed in $\Theta((n+|\mathcal{L}|)\log|\Sigma|)$ time and occupies $\Theta(|\mathcal{F}|\log|\Sigma|+|\mathcal{L}|)$ space. The construction time is further improved to $\Theta(n+|\mathcal{L}|\log|\Sigma|)$ time in [8]. Moreover, if only Query 1 is concerned, the space for the data structure can be further reduced to $\Theta(|\mathcal{F}|\log|\Sigma|)$. As $|\mathcal{F}|\le|\mathcal{L}|\le n|\Sigma|$ [7], the above query results are the best known ones so far. All previous algorithms used the naming technique for the indexing problem.

In this paper, we focus on the indexing problem of Query 1 and Query 2. All these previous results for queries are independent of the size of the input set C. Amir *et al.* [1] asked if there is a solution for Query 1 in $O(|C|\times \text{polylog}\, n)$ time. We answer this open problem by designing a new data structure, called the *lexi-string trie*, for answering Query 1 and Query 2 in $O(|C|\log(|\Sigma|/|C|))$ time and $O(|C|\log(|\Sigma|/|C|)+K)$ time, respectively. Note that the term $|C|\log(|\Sigma|/|C|)$ is equal to $\Theta(|\Sigma|)$ in the worst case, but is usually better than $\Theta(|\Sigma|)$. The construction of this data structure takes $\Theta(n+|\mathcal{L}|\log|\Sigma|)$ time by using the

Table 1. The results of the text fingerprinting problem

	Construction Time	Storage	Query Times
[1,3]	$\Theta(n\|\Sigma\|\log\|\Sigma\|)$	$\Theta(n\|\Sigma\|\log\|\Sigma\|)$	Q1: $O(\|\Sigma\|\log n)$
			Q2: $O(\|\Sigma\|\log n+K)$
[7,8]	$\Theta(n+\|\mathcal{L}\|\log\|\Sigma\|)$	Q1: $\Theta(\|\mathcal{F}\|\log\|\Sigma\|)$	Q1: $\Theta(\|\Sigma\|)$
		Q2: $\Theta(\|\mathcal{F}\|\log\|\Sigma\|+\|\mathcal{L}\|)$	Q2: $\Theta(\|\Sigma\|+K)$
Ours	$\Theta(n+\|\mathcal{L}\|\log\|\Sigma\|)$	Q1: $\Theta(\|\mathcal{F}\|)$	Q1: $O(\|C\|\log(\|\Sigma\|/\|C\|))$
		Q2: $\Theta(\|\mathcal{L}\|)$	Q2: $O(\|C\|\log(\|\Sigma\|/\|C\|)+K)$

Find_All algorithm of Kolpakov and Raffinot as a tool. During the construction, $\Theta(|\mathcal{F}| \log |\Sigma| + |\mathcal{L}|)$ working space is needed. On the other hand, the lexi-string trie occupies only $\Theta(|\mathcal{F}|)$ space for Query 1 and occupies $\Theta(|\mathcal{L}|)$ space for Query 2, which is more space-efficient than previous results. Instead of applying the naming technique, our query algorithm uses a new approach by converting the input set C into a string, reducing the fingerprint query into a pattern matching query, and then solving the latter query by the lexi-string trie. Table 1 summarizes the comparison of results.

The rest of this paper is organized as follows. Next section gives notation and preliminaries. In Section 3, we review the previous result of Kolpakov and Raffinot. In Section 4, we describe the lexi-string trie and show how to use it to answer the fingerprint queries, while in Section 5, we show its construction. Finally, Section 6 remarks a further improved result which we can obtain.

2 Notation and Preliminaries

Let $S = s_1 s_2 \dots s_n$ be a string of length n over a finite, ordered alphabet $\Sigma = \{1, 2, \dots, |\Sigma|\}$. Let $S[i, j]$ be the substring $s_i \dots s_j$ of S, where $1 \leq i \leq j \leq n$. The *fingerprint* of $S[i, j]$ is the set of distinct characters appearing in $S[i, j]$, which is denoted by $C_S(i, j)$. Let $\mathcal{F}$ denote the set of all distinct fingerprints of all substrings of a string S. For convenience, let $\emptyset$ (empty set) belong to $\mathcal{F}$.

Given a fingerprint F, an interval $[i, j]$ is a *location* of F in S if and only if $C_S(i, j) = F$. An interval $[i, j]$ is a *maximal location* of F in S if and only if $C_S(i, j) = F$ and $s_{i-1} \notin F, s_{j+1} \notin F$, where s_0 and s_{n+1} are assumed to be 0. Let $\mathcal{L}$ be the set of all maximal locations of all fingerprints. In this paper, we consider constructing an index on S to support the following queries:

Query 1: Given a set $C \subseteq \Sigma$, answer if $C \in \mathcal{F}$.
Query 2: Given a set $C \subseteq \Sigma$, find all maximal locations of C in S.

Given a string S, let S' be a string obtained by replacing each maximal substrings of repeated characters c^k in S by a single character c. We call S' the *simple format* of S. Kolpakov and Raffinot showed the following lemma.

Lemma 1. [8] *In $O(n)$ time, we can build an $O(n)$-space index such that each fingerprint query on S can be reduced to a fingerprint query on S' in $O(1)$ time.*

Based on Lemma 1, to construct an index for a string S that supports fingerprint query, we can focus on constructing an index for its simple format instead. So, in the following, we assume that S is already in a simple format. It follows that each interval $[i, i]$ will be a maximal location in S. Therefore, $n \leq |\mathcal{L}|$.

3 The Fingerprint Tree

In this section, we briefly review the fingerprint tree proposed by Kolpakov and Raffinot [7,8] for the text fingerprinting problem; at the same time, we extract

the useful concepts that are needed in later sections. In the following, we first describe a preprocessing phase, which is used to compute the information for generating the fingerprint tree. Then, we give the definition of the fingerprint tree and explain how to use it to answer the desired queries.

3.1 The Preprocessing Phase

The main goals of the preprocessing phase are (1) to find all maximal locations in S, and (2) to find all distinct fingerprints in S, and (3) for each distinct fingerprint F, to find all maximal locations whose fingerprint is F.

To find all maximal locations, Kolpakov and Raffinot [7] showed that this can be done in $\Theta(|\mathcal{L}|)$ time. In fact, they gave a stronger result:

Lemma 2. [7] *Let $maxloc_i = (i_1, i_2, \ldots, i_k)$ be a list of sorted positions, with $i = i_1 < i_2 < \ldots < i_k$, such that $[i, i_1], [i, i_2], \ldots, [i, i_k]$ are exactly all maximal locations in S with starting position i. Then, all maximal location lists $maxloc_i, i = 1, 2, \ldots, n$, can be computed in $\Theta(|\mathcal{L}|)$ time.*

Next, we want to find all distinct fingerprints in S. However, instead of outputting the fingerprints explicitly, we will just assign a name for each distinct fingerprint. (Later, we will discuss how to retrieve the actual fingerprint when it is needed, based on the name.) The name assignment process is based on the naming technique of Amir *et al.* [1]. We begin by defining some important concepts:

1. For a fingerprint F, we define a $|\Sigma|$-bit array called *fingerprint table* B_F, such that $B_F[\alpha] = 1$ if character α is in F, and $B_F[\alpha] = 0$ otherwise.
2. The *name tree* of a fingerprint F is a complete ordered binary tree with $|\Sigma|$ leaves, satisfying the following properties:
   (a) For a node containing the $x^{\text{th}}, x+1^{\text{th}}, \ldots, y^{\text{th}}$ leaves in its subtree, the node corresponds to the contents of the subarray $B_F[x..y]$. (Thus, the j^{th} leaf corresponds to $B_F[j]$, and the root corresponds to $B_F[1..|\Sigma|]$.)
   (b) Each node has a number as its name. For the α^{th} leaf, it is named with the content of $B_F[\alpha]$. For the internal nodes, they are named *consistently* such that two nodes have the same name if and only if the contents of their corresponding subarrays are the same.
   (c) The name of the root represents the whole B_F, and is called the *fingerprint name* of F.
3. If some internal node has the name x and its left and right children have names y and z, respectively, the ordered pair of names (y, z) is then called the *source pair* of the name x.
4. Let $T_1, T_2, \ldots, T_k$ be a set of name trees, where T_i is based on some fingerprint F_i. We say the set of name trees is *agreeing* if the following holds:
   (a) For any two nodes from two distinct trees, they have the same name if and only if the contents of their corresponding subarrays are the same.

By the above definition, given an agreeing set of name trees built on the fingerprints of all maximal locations $\in \mathcal{L}$, two maximal locations have the same fingerprint if and only if the same fingerprint names are assigned to their name tree. Thus, to find all distinct fingerprints in S, it is sufficient to build an agreeing set of name trees for the fingerprints of all maximal locations in $\mathcal{L}$. Then, we can find all distinct fingerprint names as the representatives of their fingerprints, and attach a list of maximal locations to each fingerprint name.

Since each name tree is of size $\Theta(|\Sigma|)$, building every name tree in an agreeing set explicitly may cost too much. However, if we focus on only getting the fingerprint names in some agreeing set, it can be done faster. By exploiting the fact that the fingerprints of two adjacent maximal locations in $maxloc_i$ differ by exactly one character, Kolpakov and Raffinot [8] gave the following result.

Lemma 3. [8] *We can find all distinct fingerprint names for some agreeing set and their maximal location lists in* $\Theta(|\mathcal{L}|\log|\Sigma|)$ *time and* $\Theta(|\mathcal{F}|\log|\Sigma| + |\mathcal{L}|)$ *space. In addition, the fingerprint names are sorted such that for any two fingerprint* F *and* F' *of* $\mathcal{F}$*, the fingerprint name of* F *is smaller than the fingerprint name of* F' *if and only if* B_F *is lexicographically smaller than* $B_{F'}$*.*

Actually, Kolpakov and Raffinot's algorithm is based on computing an implicit representation of an agreeing set $\boldsymbol{R}$, which uses only $\Theta(|\mathcal{F}|\log|\Sigma|)$ distinct names on all name trees. Every name in $\boldsymbol{R}$ and its source pairs can be computed in total $\Theta(|\mathcal{L}|\log|\Sigma|)$ time and $\Theta(|\mathcal{F}|\log|\Sigma| + |\mathcal{L}|)$ space. With these information, it can be observed that any name tree in $\boldsymbol{R}$ can be decoded from its fingerprint name by recursively extracting the source pairs.

3.2 The Definition of Fingerprint Tree and Its Searching Algorithm

The fingerprint tree [7,8] is a compacted binary tree containing the bitstrings B_F for all fingerprints $F \in \mathcal{F}$. Each leaf in the tree corresponds to some fingerprint F in S, and it stores an additional pointer to the list of maximal locations whose fingerprint is F. Each edge in the tree thus corresponds to a subarray of some fingerprint table. Instead of labeling the edge with the bitstring of the subarray explicitly, Kolpakov and Raffinot showed that we can encode this bitstring, say b, with two appropriate names from the agreeing set $\boldsymbol{R}$, so that with the help of the source pairs, the bitstring b can be decoded in $\Theta(|b|)$ time. Thus, the fingerprint tree occupies $\Theta(|\mathcal{F}|)$ space.

When a query C comes, we construct the fingerprint table for C, and traverse the fingerprint tree to check whether C is a fingerprint in S. As the bitstring of each edge can be retrieved in time proportional to its length, the process takes $\Theta(|\Sigma|)$ time. This gives the following theorem.

Theorem 1. [8] *The fingerprint tree (and the source pairs) can be constructed in* $\Theta(|\mathcal{L}|\log|\Sigma|)$ *time and* $\Theta(|\mathcal{F}|\log|\Sigma|)$ *space, which supports Query 1 in* $\Theta(|\Sigma|)$ *time and Query 2 in* $\Theta(|\Sigma|+K)$ *time, where* K *is the number of maximal locations of* C*.*

4 The Improved Query Algorithm

In the following two sections, a different approach is proposed for indexing all fingerprints in $\mathcal{F}$. More specifically, the fingerprints in $\mathcal{F}$ are treated as strings and indexed by a compacted data structure, called the *lexi-string trie*. Given a query set $C \subseteq \Sigma$, instead of encoding C by the naming technique, we convert C into a string and find its match in the lexi-string trie. In Subsection 4.1, we give the definition of the lexi-string trie. Then, in Subsection 4.2, the algorithms for answering Query 1 and Query 2 by the lexi-string trie are discussed. The construction of the data structure is discussed in Section 5.

4.1 The Definition of the Lexi-String Trie

In the previous algorithms [1,3,7,8], each query C is viewed as an array of size $|\Sigma|$, so the query time has a trivial $\Omega(|\Sigma|)$ lower bound. For the case where a query C is given by a set of distinct characters, Amir [1] asked if it is possible to process the query in $O(|C| \times \text{polylog } n)$ time. In this section, we answer the open problem by an intuitive idea: treat each query and fingerprint as a string such that we can apply string handling techniques. For this purpose, we need some definitions. The operators $<_L$ and $>_L$ are used to represent the lexicographical order among strings. Also, an *end-of-string* symbol $\$ \notin \Sigma$ is introduced such that $\$ >_L c$ for each $c \in \Sigma$, and let $\Sigma' = \Sigma \cup \{\$\}$. For any character set $C \subseteq \Sigma$, the *lexi-string* of C, which is denoted by $LS(C)$, is defined to be the string formed by concatenating all the characters in C and the symbol \$ according to the $<_L$ order. For example, assuming alphabetical order, the lexi-strings of the sets C_1 = {a, b, d, f} and C_2 = {d, b, a, f} are both "abdf\$".

In this section, we focus first on Query 1: given a query set $C \subseteq \Sigma$, determine whether $C \in \mathcal{F}$. Obviously, $C \in \mathcal{F}$ if and only if there exists some fingerprint $F \in \mathcal{F}$ such that $LS(F) = LS(C)$. Let Z be the collection of all lexi-strings $LS(F)$ of all fingerprints $F \in \mathcal{F}$. The query can thus be answered by determining whether there is an exact match for $LS(C)$ in the collection Z. By this observation, the concept of text indexing can be applied to solving the problem. In fact, we aim to construct a space-efficient data structure to implicitly represent Z, which supports the match query for any given $LS(C)$.

We begin from the definition of the *Patricia trie* [6], a classical text indexing data structure. Let Y be a collection of distinct strings. The Patricia trie built on Y can be defined in three steps. First, a compacted trie is built for Y. Next, each node in the compacted trie is labeled with the length of the string represented by this node. This value is called the *skip value*. Finally, the substring labeled on each compacted trie edge is replaced by its first character, called the *branching character*. The key property is, for any two leaves, the string represented by their lowest common ancestor is the longest common prefix of the two strings represented by this two leaves. Since the Patricia trie does not explicitly store the substrings on the edges, its storage size is proportional to only the number of strings in Y.

Now we can define the main data structure. Given the text S, the lexi-string trie T of S, which is abbreviated as the *LS trie*, is defined to be the Patricia trie built upon the lexi-string collection Z with respect to the fingerprint collection $\mathcal{F}$ of S. Inherited from the compacted trie and the Patricia trie, the LS trie T has the following basic properties:

1. T has $|\mathcal{F}|$ leaves and total $\Theta(|\mathcal{F}|)$ nodes.
2. Each leaf of T represents exactly a lexi-string $LS(F)$ of some fingerprint $F \in \mathcal{F}$.
3. Each internal node of T has at most $|\Sigma'|$ branches, which are labeled with distinct branching characters $\in \Sigma'$ and ordered by these characters according to the $<_L$ order.

Furthermore, a constant-size label, called the *recognizer*, is also stored at each leaf of T, which is used to retrieve the characters belonging to the fingerprint associated with this leaf. The usage of the recognizer will be described in Subsection 4.2.2. Since each node and edge costs only $O(1)$ space, the LS trie requires only $\Theta(|\mathcal{F}|)$ space in total.

4.2 Answering the Queries

For any given query $C \subseteq \Sigma$, we now show how to determine whether $LS(C) \in Z$ by the LS trie T. This is done in two phases. At first, by performing an $O(|C| \log(|\Sigma|/|C|))$-time traversal on T, a candidate lexi-string $LS(F)$ is retrieved from the collection Z. Then, the equivalence between $LS(C)$ and $LS(F)$ is verified in $O(|C|)$ time. The two phases are described in 4.2.1 and 4.2.2, respectively. For ease of discussion, the characters in C are assumed to be given according to the $<_L$ order, such that $LS(C)$ is trivially created. (The assumption can be easily resolved, as described in 4.2.2).

4.2.1 Finding a Candidate Lexi-String

Given a lexi-string $LS(C)$, consider the problem of finding its exact match in Z by the LS trie T. In general, there are two ways to perform search in a Patricia trie. One way is to record a pointer on each edge of the Patricia trie during construction. Each pointer points to one position of one of the strings in the collection, such that the substring associated with this edge can be retrieved directly from the string. Thus, it is just like searching on a compacted trie. An example of such approach is the famous *suffix tree* [10], which is a variation of the Patricia trie. The fingerprint tree used in [7,8] can also be seen as a Patricia binary trie. However, due to space consideration, the lexi-string collection Z is not maintained explicitly for the LS trie. Therefore, this approach is not suitable for searching in the LS trie.

Instead, we use the other traditional approach [4,6] for the LS trie T. The approach is called the *blind search* [4]. Given the lexi-string $LS(C)$, the blind search is performed in two phases: the traversal phase and the verification phase. In the traversal phase, we traverse on T downward by $LS(C)$ until a leaf is reached. The traversal begins from the root. Let $skip(u)$ denote the skip value

labeled on the node $u \in T$. When a node u is visited in the traversal, the first $skip(u)$ characters in $LS(C)$ are all ignored, and the downward branch is selected by comparing the $skip(u) + 1^{\text{th}}$ character among the branching characters of the branches of u. This procedure terminates in three cases: (1) a leaf is reached; (2) there exists no branch with the same branching character; (3) no sufficient characters in $LS(C)$ to continue the traversal, i.e., the length of $LS(C) \leq skip(u)$ for some interval node u. For the two latter cases, we claim that the blind search is *failed* and $C \notin \mathcal{F}$. If a leaf v is reached, the length of $LS(C)$ is then compared with $skip(v)$, which is the length of the lexi-string represented by v. The blind search is also claimed to be failed if they are not equal. Otherwise, we enter the verification phase to determine whether $LS(C)$ is equal to the lexi-string represented by v. The blind search is said to be *successful* when the two lexi-strings are equal. For the sake of clarity, the details about the verification phase will be described in Subsection 4.2.2. Suppose the verification phase correctly determines the equivalence between them, we have the following essential lemma. Due to page limit, the proof is omitted.

Lemma 4. *$LS(C)$ is in the collection Z if and only if the blind search procedure is successful on T.*

Corollary 5. *C is a fingerprint in S if and only if the blind search procedure is successful.*

After describing the concept of the blind search, we proceed to discuss its time complexity on the LS trie T. Consider first the traversal phase. At each node being visited, the traversal tests the skip value and, if not failed, selects an appropriate branch to continue. The traversal does at most $|C| + 1$ times of branch selections. Recall that there are at most $|\Sigma'| = |\Sigma| + 1$ branches on each internal node of T and they are ordered by their branching characters. Therefore, by simply performing binary search for each branch selection, we have the time complexity $O(|C| \log |\Sigma|)$ for the traversal phase. However, this time may exceed $\Theta(|\Sigma|)$ in the worst case. In the following, we show how to do better in branch selection.

The concept is to search the appropriate branch by the exponential search. Suppose the traversal goes through the nodes $v_1, v_2, \ldots, v_k$. While selecting on a node v_i, the branching characters on the $1^{\text{st}}, 2^{\text{nd}}, 4^{\text{th}}, 8^{\text{th}}, \ldots$ branches are checked one by one, until we find some j where the appropriate branch is between the $2^j + 1^{\text{th}}$ and the $2^{j+1\text{th}}$ branches. The binary search is then applied to the 2^j branches for finding the appropriate branch. Suppose that the ${d_i}^{\text{th}}$ branch is the appropriate one for the node v_i, obviously, it will be found in at most $2\lceil \log d_i \rceil$ steps. With some efforts, the following result can be obtained, which is bounded by $\Theta(|\Sigma|)$. The proof is omitted due to page limit.

Lemma 6. *The traversal phase requires $O(|C| \log(|\Sigma|/|C|))$ time.*

4.2.2 Verifying the Candidate Lexi-String

In the previous subsection, we introduce the blind search procedure on the LS trie and discuss its traversal phase. It remains to implement its verification phase.

Suppose the traversal reaches a leaf of T, which represents the lexi-string $LS(F)$ of some fingerprint $F \in \mathcal{F}$. The objective of the verification phase is to determine whether $LS(C) = LS(F)$ or, equivalently, whether $F = C$. This is not trivial because neither F nor $LS(F)$ is explicitly maintained. In this subsection, we propose an auxiliary data structure such that the characters of any specified fingerprint F can be retrieved in $\Theta(|F|)$ time. Then, we discuss how to verify C with these characters. The concept of the data structure is based on the following key lemma. The proof is omitted due to page limit.

Lemma 7. *For any non-empty fingerprint $F \in \mathcal{F}$, there exist at least a fingerprint $F' \in \mathcal{F}$ such that $F' = F \backslash \{\alpha\}$ for some character $\alpha \in F$.*

It follows that we can get the characters in F by applying Lemma 7 recursively on F and collecting the character α. This idea can be realized by defining the following auxiliary data structure. First, $|\mathcal{F}|$ nodes are created for the $|\mathcal{F}|$ fingerprints in $\mathcal{F}$. Let $v(F)$ denote the node which corresponds to the fingerprint F. For each $F \in \mathcal{F}$ except $\emptyset$, an arbitrary fingerprint F' is selected, where $F' = F \backslash \{\alpha\}$ for some character $\alpha \in F$. Then, an edge labeled with the character α is created, which directs from the node $v(F)$ to the node $v(F')$. It is easy to see that these edges connect the $|\mathcal{F}|$ nodes and form a directed tree rooted at the node $v(\emptyset)$. This is called the *backtracking tree*, which requires total $O(|\mathcal{F}|)$ space. On this tree, the character of a fingerprint F can be retrieved in $\Theta(|F|)$ time by traversing the path from $v(F)$ toward $v(\emptyset)$ and listing the characters on the path. Recall that each leaf of the LS trie has to store a recognizer for its retrieval. Therefore, for each $F \in \mathcal{F}$, the pointer to $v(F)$ is used as its recognizer.

Lemma 8. *For any fingerprint F specified by its recognizer, the characters in F can be retrieved in $\Theta(|F|)$ time by the backtracking tree.*

Consider first the verification phase for a query C. A leaf v must be reached at the end of the traversal phase. Suppose F is the fingerprint associated with the leaf v. The checking in the traversal phase guarantees that $|F| = |C|$. Thus, the characters in F is retrieved in $O(|C|)$ time. Note that the retrieved characters are not sorted in lexicographical order. To check the equivalence between F and C, one way is to sort these characters and compare $LS(F)$ with $LS(C)$. Generally, sorting $|C|$ characters takes $O(|C| \log |C|)$ time. However, the time complexity can be improved by a simple bucket sorting. First, the alphabet space $[1, |\Sigma|]$ is divided into $|C|$ equal-sized buckets, and the characters in F and C are distributed into the buckets. Then, each bucket runs a sorting. It is easy to see that the sorting takes total $O(|C| \log(|\Sigma|/|C|))$ time. Therefore, the equivalence between F and C is determined in $O(|C| \log(|\Sigma|/|C|))$ time. Note that this sorting can also be used to resolve the sorted assumption on C.

Lemma 9. *The verification phase can be done in $O(|C| \log(|\Sigma|/|C|))$ time.*

Now we show how to construct the backtracking tree in the preprocessing. By Lemma 3, we have computed $|\mathcal{F}|$ fingerprint names to represent the fingerprints in $\mathcal{F}$ and the maximal location list $maxloc_i$ for each i. Also, each fingerprint

name is assigned with a list of locations, which corresponds to the entries in the n maximal location lists. Note that, for any two consecutive locations in a maximal location list, their corresponding fingerprints differ by exactly one character. By this fact, it is easy to build the backtracking tree in $\Theta(|\mathcal{F}|)$ time according to the steps in the definition. Due to page limit, the details are omitted.

Lemma 10. *The backtracking tree can be constructed in* $\Theta(|\mathcal{F}|)$ *time and space.*

By Corollary 5, Lemmas 6, 9, and 10, the following theorem is derived.

Theorem 2. *Query 1 can be done in* $O(|C| \log(|\Sigma|/|C|))$ *time using* $\Theta(|\mathcal{F}|)$ *storage.*

For answering Query 2, we can simply attach the list of maximal locations of a fingerprint F to the leaf corresponding to $LS(F)$ in T. The storage of these lists takes $\Theta(|\mathcal{L}|)$ space in total.

Theorem 3. *Query 2 can be done in* $O(|C| \log(|\Sigma|/|C|) + K)$ *time using* $\Theta(|\mathcal{L}|)$ *storage, where* K *is the number of maximal locations of* C*.*

5 The Construction of the LS Trie

In Section 4, we introduced the LS trie T and discussed the blind search procedure along with its auxiliary data structure. The remaining problem is the construction of T. In this section, we show a $\Theta(|\mathcal{L}| \log |\Sigma|)$-time construction for T. The basic concept is to insert a leaf for each $LS(F) \in Z$ one by one into T. To implement this idea without explicitly maintaining the collection Z, more discussions are required. By the preprocessing in Kolpakov and Raffinot's algorithm, we obtain $|\mathcal{F}|$ fingerprint names which correspond to the fingerprints in $\mathcal{F}$. Let X be the set of the $|\mathcal{F}|$ fingerprint names. For each $x \in X$, let $B(x)$ be its corresponding fingerprint table, which can also be treated as a bitstring of length $|\Sigma|$. As mentioned in Lemma 3, the preprocessing numbers the fingerprint names according to the increasing lexicographical order of their fingerprint tables. That is, for any two distinct $x, x' \in X, x < x'$ if and only $B(x) <_L B(x')$. Let $F(x)$ be the fingerprint represented by x, and $LS(x)$ be the abbreviation of the lexi-string $LS(F(x))$. We can derive a key lemma below. Due to page limit, the proof is omitted.

Lemma 11. *For any two distinct fingerprint names* $x, x' \in X$*, if* $x < x'$*,* $LS(x) >_L LS(x')$*.*

This key lemma implies that listing the fingerprint names in X in increasing order implicitly represents a lexicographical decreasing ordering of all lexi-strings in the collection Z. By definition of T, the branches of each internal node of T are ordered by their branching characters, which means that the leaves of T are ordered by the lexicographical order of their corresponding lexi-string in Z. Therefore, by scanning the names $\in X$ in decreasing order, it is as if the

leaves of T are traversed from left to right. From these arguments, we have the idea to construct T by simulating an inorder depth-first traversal on T as if T is available and then creating each node being visited accordingly. Suppose that the names in X are $x_{|\mathcal{F}|}, x_{|\mathcal{F}|-1}, \ldots, x_1$, listed in decreasing order. For each $x_i \in X$, let $v(x_i)$ be the leaf in T corresponding to $LS(x_i)$. In the beginning, we create the root of T, and "visit"the leftmost leaf $v(x_{|\mathcal{F}|})$ by creating it and its edge to the root. Suppose that $v(x_{|\mathcal{F}|}), \ldots, v(x_i)$ have been visited and we are to visit the leaf $v(x_{i-1})$. We then "backtrack" on the path from $v(x_i)$ to the root in order to "find" the internal node u branching to the next leaf $v(x_{i-1})$. Afterwards, the internal node u and the leaf $v(x_{i-1})$ are explicitly created and connected by an edge. This procedure repeats until the rightmost leaf $v(x_1)$ is inserted into T. In the following, we inspect more details inside this concept.

For any two names $x_i, x_j \in X$, let $LCP(i,j)$ be the longest common prefix of $LS(x_i)$ and $LS(x_j)$ and $lcp(i,j) = |LCP(i,j)|$ be its length. By the definition of the LS trie (a Patricia trie), the position of the branching node u between the leaves $v(x_i)$ and $v(x_{i-1})$ is closely related to $lcp(i, i-1)$. To be more specific, the branching node u for $v(x_{i-1})$ should have its skip value $skip(u) = lcp(i, i-1)$ by definition. Furthermore, when we find the position of u on the path from $v(x_i)$ to the root, either u is an existing node on the path, or u must be placed on an edge (u_1, u_2) where $skip(u_1) < skip(u) < skip(u_2)$. After the position of u is found, it remains to determine the branching characters on the branches of u toward $v(x_i)$ and $v(x_{i-1})$. By definition, the branching character from u toward $v(x_i)$ is the character in $LS(x_i)$ just after the prefix $LCP(i, i-1)$. A similar argument applies for the branching character from u to $v(x_{i-1})$. All these pieces of information require the LCP computation between lexi-strings.

For any $x \in X$, it is obvious that the lexi-string $LS(x)$ can be derived directly by finding each occurrence of 1 in the fingerprint table $B(x)$ from beginning and listing its corresponding character. For any two x_i, $x_j \in X$, let $LCP_B(i,j)$ be the longest common prefix of $B(x_i)$ and $B(x_j)$ and $lcp_B(i,j) = |LCP_B(i,j)|$ be its length. From the above argument, $LCP(i,j)$ can be derived by finding the 1s in $LCP_B(i,j)$, and $lcp(i,j)$ is the number of 1s in $LCP_B(i,j)$. Furthermore, by using such mapping, the character in $LS(x_i)$ just after $LCP(i,j)$ can be retrieved by finding the first occurrence of 1 in $B(x_i)$'s suffix starting at $lcp_B(i,j)+1$. Note that if no 1 exists in the suffix, the desired character would be \$. The same argument holds for the case of $LS(x_j)$.

It is easy to compute $LCP_B(i,j)$ by using the source pairs derived in Kolpakov and Raffinot's preprocessing as follows. Suppose (y_i, z_i) be the source pair of x_i and (y_j, z_j) be the source pair of x_j. We first compare y_i with y_j. If $y_i \neq y_j$, we know that $LCP_B(i,j)$ must be equal to the longest common prefix of $B(y_i)$ and $B(y_j)$, where $B(y)$ denote the bitstring corresponding to a non-fingerprint name y. Otherwise, $LCP_B(i,j)$ should be the concatenation of $B(y_i)$ (or $B(y_j)$) and the longest common prefix of $B(z_i)$ and $B(z_j)$. In either case, the desired longest common prefix can be obtained by recursively comparing their source pairs. In this way, we can find $q \leq O(\log|\Sigma|)$ names in $O(\log|\Sigma|)$ time, such that the concatenation of their corresponding bitstrings is $LCP_B(i,j)$.

With a simple preprocessing, $lcp(i, j)$ can be computed from these names. For each name y, let $num_1(y)$ be the number of 1s in $B(y)$. Since the source pairs of all names are computed, it is easy to obtain $num_1(y)$ for all y in $O(|\mathcal{F}| \log |\Sigma|)$ time by summing up in a bottom-up way. Thus, $lcp(i, j)$ can be trivially derived from the q names. Also, in a similar way, we can find $q' \leq O(\log |\Sigma|)$ names for the suffix of $B(x_i)$ starting at $lcp_B(i, j) + 1$. To find the first occurrence of 1 in the suffix, we only need to find the first y in the q' names where $num_1(y) > 0$ and retrieve the first 1 in $B(y)$ in a similar recursive way. Same arguments applies to the searching in the suffix of $B(x_j)$. Thus, each of these LCP related operations take $O(\log |\Sigma|)$ time.

From the above discussion, while inserting a leaf $v(x_{i-1})$ into T, it takes a total of $O(\log |\Sigma|)$ time to compute the skip value and the branching characters of the branching node u. Then, the position of u can be found in $O(\log |\Sigma|)$ time by applying binary search on the path from $v(x_i)$ to the root. Thus, it requires a total of $O(|\mathcal{F}| \log |\Sigma|)$ time to compute these values for all leaves. Also, it is easy to see that the corresponding recognizer can be assigned to each leaf in $O(1)$ time. Finally, the preprocessing of Kolpakov and Raffinot requires $\Theta(|\mathcal{L}| \log |\Sigma|)$ time and $\Theta(|\mathcal{F}| \log |\Sigma| + |\mathcal{L}|)$ space. We have the following result.

Theorem 4. *Using $\Theta(|\mathcal{L}| \log |\Sigma|)$ time and $\Theta(|\mathcal{F}| \log |\Sigma| + |\mathcal{L}|)$ space, we can construct the LS trie T, which uses only $\Theta(|\mathcal{F}|)$ space.*

6 Further Improvement

By observing some new properties and with minor modifications, we can reduce the space requirement of the construction to $O(|\mathcal{L}|)$ and create a modified LS trie T' without increasing the construction time. The modified LS trie T' can still answer Query 1 and Query 2 correctly in the same time. Thus, we claim the following result. Due to space limit, the proof is omitted here.

Theorem 5. *Using $\Theta(|\mathcal{L}| \log |\Sigma|)$ time and $\Theta(|\mathcal{L}|)$ space, we can construct a modified LS trie T'. Based on T', we can answer Query 1 in $O(|C| \log(|\Sigma|/|C|))$ time with $\Theta(|\mathcal{F}|)$ space, and answer Query 2 in $O(|C| \log(|\Sigma|/|C|) + K)$ time with $\Theta(|\mathcal{L}|)$ space, where K is the number of maximal locations of C.*

References

1. Amir, A., Apostolico, A., Landau, G.M., Satta, G.: Efficient text fingerprinting via Parikh mapping. Journal of Discrete Algorithms 1, 409–421 (2003)
2. Dandekar, T., Snel, B., Huynen, M., Bork, P.: Conservation of gene order: a fingerprint of proteins that physically interact. Trends in biochemical sciences 23(9), 324–328 (1998)
3. Didier, G., Schmidt, T., Stoye, J., Tsur, D.: Character sets of strings. Journal of Discrete Algorithms 5(2), 330–340 (2007)
4. Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications. Journal of the ACM 46(2), 236–280 (1999)

5. Karlsson, F., Voutilainen, A., Heikkilä, J., Anttila, A.: Constraint grammar: A language-independent system for parsing unrestricted text. de Gruyter, Berlin (1995)
6. Knuth, D.E.: The art of computer programming. In: Sorting and searching, vol. 3, Addison-Wesley, London, UK (1973)
7. Kolpakov, R., Raffinot, M.: New algorithms for text fingerprinting. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 342–353. Springer, Heidelberg (2006)
8. Kolpakov, R., Raffinot, M.: New algorithms for text fingerprinting (2006) (unpublished, submitted), http://www-igm.univ-mlv.fr/~raffinot/ftp/fingerprint.pdf
9. Rogozin, I.B., Makarova, K.S., Murvai, J., Czabarka, E., Wolf, Y.I., Tatusov, R.L., Szekely, L.A., Koonin, E.V.: Connected gene neighborhoods in prokaryotic genomes. Nucleic Acids Research 30(10), 2212–2223 (2002)
10. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory, pp. 1–11 (1973)

Polynomial Time Algorithms for Minimum Energy Scheduling

Philippe Baptiste[1,⋆], Marek Chrobak[2,⋆⋆], and Christoph Dürr[1,⋆]

[1] CNRS, LIX UMR 7161, Ecole Polytechnique 91128 Palaiseau, France
[2] Department of Computer Science, University of California, Riverside, CA 92521, USA

Abstract. The aim of power management policies is to reduce the amount of energy consumed by computer systems while maintaining satisfactory level of performance. One common method for saving energy is to simply suspend the system during the idle times. No energy is consumed in the suspend mode. However, the process of waking up the system itself requires a certain fixed amount of energy, and thus suspending the system is beneficial only if the idle time is long enough to compensate for this additional energy expenditure. In the specific problem studied in the paper, we have a set of jobs with release times and deadlines that need to be executed on a single processor. Preemptions are allowed. The processor requires energy L to be woken up and, when it is on, it uses the energy at a rate of R units per unit of time. It has been an open problem whether a schedule minimizing the overall energy consumption can be computed in polynomial time. We solve this problem in positive, by providing an $O(n^5)$-time algorithm. In addition we provide an $O(n^4)$-time algorithm for computing the minimum energy schedule when all jobs have unit length.

1 Introduction

Power Management Strategies. The aim of power management policies is to reduce the amount of energy consumed by computer systems while maintaining satisfactory level of performance. One common method for saving energy is a *power-down mechanism*, which is to simply suspend the system during the idle times. The amount of energy used in the suspend mode is negligible. However, during the wake-up process the system requires a certain fixed amount of *start-up* energy, and thus suspending the system is beneficial only if the idle time is long enough to compensate for this extra energy expenditure. The intuition is that we can reduce energy consumption if we schedule the work to performed so that we reduce the weighted sum of two quantities: the total number of busy periods and the total length of "short" idle periods, when the system is left on.

Scheduling to Minimize Energy Consumption. The scheduling problem we study in this paper is quite fundamental. We are given a set of jobs with release

⋆ Supported by CNRS/NSF grant 17171 and ANR Alpage.
⋆⋆ Supported by NSF grants OISE-0340752 and CCR-0208856.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 136–150, 2007.

times and deadlines that need to be executed on a single processor. Preemptions are allowed. The processor requires energy L to be woken up and, when it is on, it uses the energy at a rate of R units per unit of time. The objective is to compute a feasible schedule that minimizes the overall energy consumption. Denoting by E the energy consumption function, this problem can be classified using Graham's notation as $1|r_j; \text{pmtn}|E$.

The question whether this problem can be solved in polynomial time was posed by Irani and Pruhs [8], who write that "... Many seemingly more complicated problems in this area can be essentially reduced to this problem, so a polynomial time algorithm for this problem would have wide application." Some progress towards resolving this question has already been reported. Chretienne [3] proved that it is possible to decide in polynomial time whether there is a schedule with no idle time. More recently, Baptiste [2] showed that the problem can be solved in time $O(n^7)$ for unit-length jobs.

Our Results. We solve the open problem posed by Irani and Pruhs [8], by providing a polynomial-time algorithm for $1|r_j; \text{pmtn}|E$. Our algorithm is based on dynamic programming and it runs in time $O(n^5)$. Thus not only our algorithm solves a more general version of the problem, but is also faster than the algorithm for unit jobs in [2]. For the case of unit jobs (that is, $1|r_j; p_j = 1|E$), we improve the running time to $O(n^4)$.

The paper is organized as follows. First, in Section 2, we introduce the necessary terminology and establish some basic properties. Our algorithms are developed gradually in the sections that follow. We start with the special case of minimizing the number of gaps for unit jobs, that is $1|r_j; p_j = 1; L = 1|E$, for which we describe an $O(n^4)$-time algorithm in Section 3. Next, in Section 4, we extend this algorithm to jobs of arbitrary length $(1|r_j; \text{pmtn}; L = 1|E)$, increasing the running time to $O(n^5)$. Finally, in Section 5, we show how to extend these algorithms to arbitrary L without increasing their running times.

We remark that our algorithms are sensitive to the structure of the input instance and on typical instances they are likely to run significantly faster than their worst-case bounds.

Other Relevant Work. The non-preemptive version of our problem, that is $1|r_j|E$, can be easily shown to be $\mathbb{NP}$-hard in the strong sense, even for $L = 1$ (when the objective is to only minimize the number of gaps), by reduction from 3-Partition [4, problem SS1].

More sophisticated power management systems may involve several sleep states with decreasing rates of energy consumption and increasing wake-up overheads. In addition, they may also employ a method called *speed scaling* that relies on the fact that the speed (or frequency) of processors can be changed on-line. As the energy required to perform the job increases quickly with the speed of the processor, speed scaling policies tend to slow down the processor while ensuring that all jobs meet their deadlines (see [8], for example). This problem is a generalization of $1|r_j|E$ and its status remains open. A polynomial-time 2-approximation algorithm for this problem (with two power states) appeared in [6].

As jobs to be executed are often not known in advance, the on-line version of energy minimization is of significant interest. Online algorithms for power-down strategies with multiple power states were considered in [5,7,1]. In these works, however, jobs are critical, that is, they must be executed as soon as they are released, and the online algorithm only needs to determine the appropriate power-down state when the machine is idle. The work of Gupta, Irani and Shukla [6] on power-down with speed scaling is more relevant to ours, as it involves aspects of job scheduling. For the specific problem studied in our paper, $1|r_j|E$, it is easy to show that no online algorithm can have a constant competitive ratio (independent of L), even for unit jobs. We refer the reader to [8] for a detailed survey on algorithmic problems in power management.

2 Preliminaries

Minimum-Energy Scheduling. Formally, an instance of the scheduling problem $1|r_j; \text{pmtn}|E$ consists of n jobs, where each job j is specified by its processing time p_j, release time r_j and deadline d_j. We have one processor that, at each step, can be on or off. When it is on, it consumes energy at the rate of R units per time step. When it is off, it does not consume any energy. Changing the state from off to on (waking up) requires additional L units of energy. Without loss of generality, we assume that $R = 1$.

The time is discrete, and is divided into unit-length intervals $[t, t+1)$, where t is an integer, called *time slots* or *steps*. For brevity, we often refer to time step $[t, t+1)$ as *time step* t. A preemptive schedule S specifies, for each time slot, whether some job is executed at this time slot and if so, which one. Each job j must be executed for p_j time slots, and all its time slots must be within the time interval $[r_j, d_j)$.

A *block* of a schedule S is a maximal interval where S is *busy* that is, executes a job. The union of all blocks of S is called its *support*. A *gap* of S is a maximal interval where S is idle (does not execute a job). By $C_j(S)$ (or simply C_j, if S is understood from context) we denote the completion time of a job j in a schedule S. By $C_{\max}(S) = \max_j C_j(S)$ we denote the maximum completion time of any job in S. We refer to $C_{\max}(S)$ as the *completion time of schedule* S.

Since the energy used on the support of all schedules is the same, it can be subtracted from the energy function for the purpose of minimization. The resulting function $E(S)$ is the "wasted energy" (when the processor is on but idle) plus L times the number of wake-ups. Formally, this can be calculated as follows. Let $[u_1, t_1], \ldots, [u_q, t_q]$ be the set of all blocks of S, where $u_1 < t_1 < u_2 < \ldots < t_q$. Then

$$E(S) = \sum_{i=2}^{q} \min\{u_i - t_{i-1}, L\}.$$

(We do not charge for the first wake-up at time u_1, since this term is independent of the schedule.) Intuitively, this formula reflects the fact that once the support

of a schedule is given, the optimal suspension and wake-up times are easy to determine: we suspend the machine during a gap if and only if its length is more than L, for otherwise it would be cheaper to keep the processor on during the gap.

Our objective is to find a schedule S that meets all job deadlines and minimizes $E(S)$. (If there is no feasible schedule, we assume that the energy value is $+\infty$.) Note that the special case $L = 1$ corresponds to simply minimizing the number of gaps.

Simplifying Assumptions. Throughout the paper we assume that jobs are ordered according to deadlines, that is $d_1 \leq \ldots \leq d_n$. Without loss of generality, we also assume that all release times are distinct and that all deadlines are distinct. Indeed, if $r_i = r_j$ for some jobs $i < j$, since the jobs cannot start both at the same time r_i, we might as well increase by 1 the release time of j. A similar argument applies to deadlines.

To simplify the presentation, we assume that the job indexed 1 is a special job with $p_1 = 1$ and $d_1 = r_1 + 1$, that is job 1 has unit length and must be scheduled at its release time. (Otherwise we can always add such an extra job, released $L + 1$ time slots before r_1. This increases each schedule's energy by exactly L and does not affect the asymptotic running time of our algorithms).

Without loss of generality, we can also assume that the input instance is feasible. A feasible schedule corresponds to a matching between units of jobs and time slots, so Hall's theorem gives us the following necessary and sufficient condition for feasibility: for all times $u < v$,

$$\sum_{u \leq r_j, d_j \leq v} p_j \leq v - u. \tag{1}$$

We can also restrict our attention to schedules S that satisfy the following *earliest-deadline property*: at any time t, either S is idle at t or it schedules a pending job with the earliest deadline. In other words, once the support of S is fixed, the jobs in the support are scheduled according to the earliest deadline policy. Using the standard exchange argument, any schedule can be converted into one that satisfies the earliest-deadline property and has the same support.

(k, s)-Schedules. We will consider certain partial schedules, that is schedules that execute only some jobs from the instance. For jobs k and s, a partial schedule S is called a *(k, s)-schedule* if it schedules all jobs $j \leq k$ with $r_s \leq r_j < C_{\max}(S)$ (recall that $C_{\max}(S)$ denotes the completion time of schedule S). From now on, unless ambiguity arises, we will omit the term "partial" and refer to partial schedules simply as schedules. When we say that that a (k, s)-schedule S has g gaps, in addition to the gaps between the blocks we also count the gap (if any) between r_s and the first block of S. For any k, s, the empty schedule is also considered to be a (k, s)-schedule. The completion time of an empty (k, s)-schedule is artificially set to r_s. (Note that, in this convention, empty (k, s)-schedules, for difference choices of k, s, are considered to be different schedules.)

The following "compression lemma" will be useful in some proofs.

Lemma 1. *Let Q be a (k,s)-schedule with $C_{\max}(Q) = u$, and let R be a (k,s) schedule with $C_{\max}(R) = v > u$ and at most g gaps. Suppose that there is a time t, $u < t \leq v$, such that there are no jobs $i \leq k$ with $u \leq r_i < t$, and that R executes some job $m < k$ with $r_m \leq u$ at or after time t. Then there is a (k,s)-schedule R' with completion time t and at most g gaps.*

Proof. We can assume that R has the earliest-deadline property. We convert R into R' by gradually reducing the completion time, without increasing the number of gaps.

Call a time slot z of R *fixed* if R executes some job j at time z and either $z = r_j$ or all times $r_j, r_{j+1}, ..., z-1$ are fixed as well. Let $[w,v]$ be the last block of R and let j be the job executed at time $v-1$. If $v = t$, we are done. For $v > t$ we show that we can reduce $C_{\max}(R)$ while preserving the assumptions of the lemma.

Suppose first that the slot $v-1$ is not fixed. In this case, execute the following operation Shift: for each non-fixed slot in $[w,v]$ move the job unit in this slot to the previous non-fixed slot in R. Shift reduces $C_{\max}(R)$ by 1 without increasing the number of gaps. We still need to justify that R is a feasible (k,s)-schedule. To this end, it is sufficient only to show that no job will be scheduled before its release time. Indeed, if a job i is executed at a non-fixed time z, where $w \leq z < v$, then, by definition, $z > r_i$ and there is a non-fixed slot in $[r_i, z-1]$, and therefore after Shift z will be schedule at or after r_i.

The other case is when the slot $v-1$ is fixed. In this case, we claim that there is a job l such that $w \leq r_l < v$ and each job i executed in $[r_l, v]$ satisfies $r_i \geq r_l$. This l can be found as follows. If $v - 1 = r_j$, let $l = j$. Otherwise, from all jobs executed in $[r_j, v-1]$ pick the job j' with minimum $r_{j'}$. Suppose that j' executes at v', $r_j \leq v' \leq v-1$. Since, by definition, the slot v' is fixed, we can apply this argument recursively, eventually obtaining the desired job l. We then perform the following operaiton Truncate: replace R by the segment of R in $[r_s, r_l]$. This decreases $C_{\max}(R)$ to r_l, and the new R is a feasible (k,s)-schedule, by the choice of l.

We repeat the process described above as long as $v > t$. Since the schedule at each step is a (k,s)-schedule, we end up with a (k,s)-schedule R'. Let $C_{\max}(R') = t' \leq t$. It is thus sufficient to prove that $t' = t$. Indeed, consider the last step, when $C_{\max}(R)$ decreases to t'. Operation Truncate reduces $C_{\max}(R)$ to a completion time of a job released after t, so it cannot reduce it to t'. Therefore the last operation applied must have been Shift that reduces $C_{\max}(R)$ by 1. Consequently, $t' = t$, as claimed.

The $U_{k,s,g}$ Function. For any $k = 0, ..., n$, $s = 1, ..., n$, and $g = 0, ..., n$, define $U_{k,s,g}$ as the maximum completion time of a (k,s)-schedule with at most g gaps. Our algorithms will compute the function $U_{k,s,g}$ and use it to determine a minimum energy schedule.

Clearly, $U_{k,s,g} \leq d_k$ and, for any fixed s and g, the function $k \mapsto U_{k,s,g}$ is increasing (not necessarily strictly). For all k and s, the function $g \mapsto U_{k,s,g}$

increases as well. We claim that in fact it increases strictly as long as $U_{k,s,g} < d_k$. Indeed, suppose that $U_{k,s,g} = u < d_k$ and that $U_{k,s,g}$ is realized by a (k,s)-schedule S with at most g gaps. We show that we can extend S to a schedule S' with $g+1$ gaps and $C_{\max}(S') > C_{\max}(S)$. If there is a job $j \leq k$ with $r_j \geq u$, take j to be such a job with minimum r_j. We must have $r_j > u$, since otherwise we could add j to S scheduling it at u without increasing the number of gaps, and thus contradicting the maximality of $C_{\max}(S)$. We thus obtain S' by scheduling j at r_j. The second case is when $r_j \leq u$ for all jobs $j \leq k$. In particular, $r_k < u$. We obtain S' by rescheduling k at u. (This creates an additional gap at the time slot where k was scheduled, for otherwise we would get a contradiction with the maximality of $C_{\max}(S)$).

An Outline of the Algorithms. Our algorithms are based on dynamic programming, and they can be thought of as consisting of two stages. First, we compute the table $U_{k,s,g}$, using dynamic programming. From this table we can determine the minimum number of gaps in the (complete) schedule (it is equal to the smallest g for which $U_{n,1,g} > \max_j r_j$.) The algorithm computing $U_{k,s,g}$ for unit jobs is called ALGA and the one for arbitrary length jobs is called ALGB.

In the second stage, described in Section 5 and called ALGC, we use the table $U_{k,s,g}$ to compute the minimum energy schedule. In other words, we show that the problem of computing the minimum energy reduces to computing the minimum number of gaps. This reduction, itself, involves again dynamic programming.

When presenting our algorithms, we will only show how to compute the minimum energy value. The algorithms can be modified in a straightforward way to compute the actual optimum schedule, without increasing the running time. (In fact, we explain how to construct such schedules in the correctness proofs).

3 Minimizing the Number of Gaps for Unit Jobs

In this section we give an $O(n^4)$-time algorithm for minimizing the number of gaps for unit jobs, that is for $1|r_j; p_j = 1; L = 1|E$. Recall that we assumed all release times to be different and all deadlines to be different, which implies that there is always a feasible schedule (providing that $d_j > r_j$ for all j).

As explained in the previous section, the algorithm computes the table $U_{k,s,g}$. The crucial idea here is this: Let S be a (k,s)-schedule that realizes $U_{k,s,g}$, that is S has g gaps and $C_{\max}(S) = u$ is maximized. Suppose that in S job k is scheduled at some time $t < u-1$. We show that then, without loss of generality, there is a job l released and scheduled at time $t+1$. Further, the segment of S in $[r_s, t]$ is a $(k-1,s)$-schedule with completion time t, the segment of S in $[t+1,u]$ is a $(k-1,l)$-schedule with completion time u, and the total number of gaps in these two schedules equals g. This naturally leads to a recurrence relation for $U_{k,s,g}$.

Algorithm ALGA**.** The algorithm computes all values $U_{k,s,g}$, for $k = 0, ..., n$, $s = 1, ..., n$ and $g = 0, ..., n$, using dynamic programming. The minimum number of gaps for the input instance is equal to the smallest g for which $U_{n,1,g} > \max_j r_j$.

To explain how to compute all values $U_{k,s,g}$, we give the recurrence relation. For the base case $k = 0$ we let $U_{0,s,g} \leftarrow r_s$ for all s and g. For $k \geq 1$, $U_{k,s,g}$ is defined recursively as follows:

$$U_{k,s,g} \leftarrow \max_{l<k, h \leq g} \begin{cases} U_{k-1,s,g} & \\ U_{k-1,s,g}+1 & \text{if } r_s \leq r_k \leq U_{k-1,s,g} \ \ \& \ \forall j < k \ r_j \neq U_{k-1,s,g} \\ d_k & \text{if } g > 0 \ \ \& \ \forall j < k \ r_j < U_{k-1,s,g-1} \\ U_{k-1,l,g-h} & \text{if } r_k < r_l = U_{k-1,s,h} + 1 \end{cases} \quad (2)$$

Note that only the last choice in the maximum depends on h and l. Also, as a careful reader might have noticed, the condition "$\forall j < k \ r_j \neq U_{k-1,s,g}$" in the second option is not necessary (the optimal solution will satisfy it automatically), but we include it to simplify the correctness proof.

In the remainder of this section we justify the correctness of the algorithm and analyze its running time. The first two lemmas establish the feasibility and the optimality of the values $U_{k,s,g}$ computed by Algorithm ALGA.

Lemma 2. *For any choice of indices k, s, g, there is a (k, s)-schedule $S_{k,s,g}$ with $C_{\max}(S_{k,s,g}) = U_{k,s,g}$ and at most g gaps.*

Proof. The proof is by induction on k. For $k = 0$, we take $S_{0,s,g}$ to be the empty (k, s)-schedule, which is trivially feasible and (by our convention) has completion time $r_s = U_{0,s,g}$.

Now fix some $k \geq 1$ and assume that the lemma holds for $k - 1$ and any s' and g', that is, for any s' and g' we have a schedule $S_{k-1,s',g'}$ with completion time $U_{k-1,s',g'}$. The construction of $S_{k,s,g}$ depends on which expression realizes the maximum (2).

If $U_{k,s,g} = U_{k-1,s,g}$, we simply take $S_{k,s,g} = S_{k-1,s,g}$. Since we did not choose the second option in the maximum, either $r_k < r_s$ or $r_k > U_{k-1,s,g}$. Therefore, directly from the inductive assumption, we get that $S_{k,s,g}$ is a (k, s)-schedule with completion time $U_{k,s,g}$.

If $U_{k,s,g} = U_{k-1,s,g} + 1$, $r_s \leq r_k \leq U_{k-1,s,g}$, and there is no job $j < k$ with $r_j = U_{k-1,s,g}$, let $S_{k,s,g}$ be the schedule obtained from $S_{k-1,s,g}$ by adding to it job k scheduled at time $u = U_{k-1,s,g}$. (Note that we must have $u < d_k$.) Then $S_{k,s,g}$ is a (k, s)-schedule with completion time $u + 1 = U_{k,s,g}$.

Next, suppose that $U_{k,s,g} = d_k$, $g > 0$, and $\max_{j<k} r_j < U_{k-1,s,g-1}$. Let $S_{k,s,g}$ be the schedule obtained from $S_{k-1,s,g-1}$ by adding to it job k scheduled at $d_k - 1$. The condition $\max_{j<k} r_j < U_{k-1,s,g-1}$ implies that no jobs $j < k$ are released between $U_{k-1,s,g-1}$ and $d_k - 1$. Therefore $S_{k,s,g}$ is a (k, s)-schedule with completion time $d_k = U_{k,s,g}$ and it has at most g gaps, since adding k can only add one gap to $S_{k-1,s,g-1}$.

Finally, suppose that $U_{k,s,g} = U_{k-1,l,g-h}$, for some $1 \leq l < k$, $0 \leq h \leq g$, that satisfy $r_k < r_l = U_{k-1,s,h} + 1$. The schedule $S_{k,s,g}$ is obtained by scheduling all jobs $j < k$ released between r_s and $r_l - 1$ using $S_{k-1,s,h}$, scheduling all jobs $j < k$ released between r_l and $U_{k-1,l,g-h} - 1$ using $S_{k-1,l,g-h}$, and scheduling job k at $r_l - 1$. By induction, $S_{k,s,g}$ is a (k, s)-schedule with completion time $U_{k,s,g}$ and at most g gaps.

Lemma 3. *For any choice of indices k, s, g, if Q is a (k, s)-schedule with at most g gaps then $C_{\max}(Q) \leq U_{k,s,g}$.*

Proof. The proof is by induction on k. For $k = 0$, any $(0, s)$-schedule is empty and thus has completion time r_s. For a given $k \geq 1$ assume that the lemma holds for $k-1$ and any s' and g', that is the values of $U_{k-1,s',g'}$ are indeed optimal. Let Q be a (k, s)-schedule with at most g gaps and maximum completion time u. Withouth loss of generality, we can assume that Q has the earliest-deadline property. The maximality of u implies that no job $j \leq k$ is released at time u, for otherwise we could add j to Q by scheduling it at u and thus increasing the completion time. (This property will be useful in the proof below.) We prove that $u \leq U_{k,s,g}$ by analyzing several cases.

Case 1: Q does not schedule job k. In this case Q is a $(k-1, s)$-schedule with completion time u, so, by induction, we have $u \leq U_{k-1,s,g} \leq U_{k,s,g}$. In all the remaining cases, we assume that Q schedules k. Obviously, this implies that $r_s \leq r_k < u$.

Case 2: Q schedules k as the last job and k is not the only job in its block. Let $u' = u - 1$, and define Q' to be Q restricted to the interval $[r_s, u']$. Then Q' is a $(k-1, s)$-schedule with completion time u' and at most g gaps, so $u' \leq U_{k-1,s,g}$, by induction. If $u' < U_{k-1,s,g}$ then, trivially, $u \leq U_{k-1,s,g} \leq U_{k,s,g}$. Otherwise, assume $u' = U_{k-1,s,g}$. Then, by the earliest deadline property, there is no job $j < k$ with $r_k = u'$. Thus the second condition in the maximum (2) is satisfied, so we have $u = u' + 1 = U_{k-1,s,g} + 1 \leq U_{k,s,g}$.

Case 3: Q schedules k as the last job and k is the only job in its block. If $u = r_s + 1$ then $k = s$ and the condition in the second option of (2) is satisfied, so we have $u = r_s + 1 = U_{s-1,s,g} + 1 = U_{s,s,g}$. Therefore we can assume now that $u > r_s + 1$, which, together with the case condition, implies that $g > 0$.

If $u < d_k$, we can modify Q by rescheduling k at time u, obtaining a $(k, s, u+1)$ schedule (by the assumption about Q, no job $j < k$ is released at u) with at most g gaps – contradicting the maximality of u.

By the above paragraph, we can assume that $u = d_k$. Let u' be the smallest time $u' \geq r_s$ such that Q is idle in $[u', d_k - 1]$. Then $\max_{j<k} r_j < u'$ and the segment of Q in $[r_s, u']$ is a $(k-1, s)$-schedule with at most $g - 1$ gaps, so, by induction, we get $u' \leq U_{k-1,s,g-1}$. Thus the third option in (2) applies and we get $u = d_k = U_{k,s,g}$.

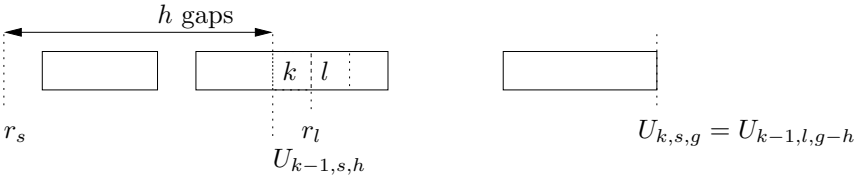


Fig. 1. Case 4 in the proof of Lemma 3

Case 4: Q schedules k and k is not the last job. Suppose that k is scheduled at time t. Note that Q is not idle at times $t-1$ and $t+1$, since otherwise we would have $u < d_k$ and we could reschedule k at u, obtaining a (k,s)-schedule with at most g gaps and completion time $u+1$, which contradicts the maximality of u. Since Q satisfies the earliest-deadline property, no job $j < k$ is pending at time t, and thus Q schedules at time $t+1$ the job $l < k$ with release time $r_l = t+1$.

Let Q_1 be the segment of Q in the interval $[r_s, t]$. Clearly, Q_1 is a $(k-1,s)$-schedule with completion time t. Denote by h the number of gaps in Q_1. We claim that Q_1 is in fact optimal, that is:

Claim 1: $t = U_{k-1,s,g}$.

Suppose for now that Claim 1 is true (see the proof below). Then the conditions of the last option in (2) are met: $l < k$, $h \le g$, and $r_k < r_l = U_{k-1,s,h}+1$. Let Q_2 be the segment of Q in $[r_l, u]$. Then Q_2 is a $(k-1,l)$-schedule with completion time u and at most $g-h$ gaps, so by induction we get $u \le U_{k-1,l,g-h}$, completing the argument for Case 4.

To complete the proof it only remains now to prove Claim 1. Denote $v = U_{k-1,s,g}$. By induction, v is the maximum completion time of a $(k-1,s)$-schedule with at most g gaps. Clearly, as Q_1 is a $(k-1,s)$-schedule with g gaps, we have $v \ge t$, and thus it suffices to show that $v \le t$. Towards contradiction, suppose that $v > t$ and let R be a $(k-1,s)$-schedule with completion time v and at most h gaps. We consider two cases.

Case (a): R schedules all jobs $j < k$ with $r_s \le r_j \le t$ in the interval $[r_s, t]$. The earliest deadline property of Q implies that there is no job $j < k$ released at time t. So R must be idle at t. We can modify Q as follows: Reschedule k at time u and replace the segment $[r_s, t+1]$ of Q by the same segment of R. Let Q' be the resulting schedule. Q' is a (k,s)-schedule. Since R has at most h gaps, there are at most h gaps in Q' in the segment $[r_s, t+1]$, so Q' has the total of at most g gaps. We thus obtain a contradiction with the choice of Q, because $C_{\max}(Q') = u+1 > C_{\max}(Q)$.

Case (b): R schedules some job $j < k$ with $r_s \le r_j \le t$ at or after t. In this case, Lemma 1 implies that there is a $(k-1,s)$-schedule R' with at most h gaps and completion time $t+1$. Replace the segment $[r_s, t+1]$ of Q by the same segment of R' and reschedule k at u. The resulting schedule Q' is a (k,s)-schedule and, since Q executes job l at time $t+1 = r_l$, Q' has at most g gaps. We thus again obtain a contradiction with the choice of Q, because $C_{\max}(Q') = u+1 > C_{\max}(Q)$.

Theorem 1. *Algorithm* ALGA *correctly computes the optimum solution for* $1|r_j; p_j = 1; L = 1|E$*, and it can be implemented in time* $O(n^4)$*.*

Proof. The correctness of Algorithm ALGA follows from Lemma 2 and Lemma 3, so it is sufficient to give the running time analysis. There are $O(n^3)$ values $U_{k,s,g}$ to be computed. For fixed k, s, g, the first two choices in the maximum (2) can be computed in time $O(1)$ and the third choice in time $O(n)$. In the last choice we maximize over pairs (l,h) that satisfy the condition $r_l = U_{k-1,s,h} + 1$, and

thus we only have $O(n)$ such pairs. Since the values of $U_{k-1,s,h}$ increase with h, we can determine all these pairs in time $O(n)$ by searching for common elements in two sorted lists: the list of release times, and the list of times $U_{k-1,s,h}+1$, for $h = 0, 1, ..., n$. Thus each value $U_{k,s,g}$ can be computed in time $O(n)$, and the overall running time is $O(n^4)$.

4 Minimizing the Number of Gaps for Arbitrary Jobs

In this section we give an $O(n^5)$-time algorithm for minimizing the number of gaps for instances with jobs of arbitrary lengths, that is for the scheduling problem $1|r_j; \text{pmtn}; L = 1|E$.

We first extend the definition of $U_{k,s,g}$ as follows. Let $0 \le k \le n$, $1 \le s \le n$, and $0 \le g \le n-1$. For any $p = 0, \ldots, p_k$, define $U_{k,s,g}(p)$ as the value of $U_{k,s,g}$ — the maximum completion time of a (k,s)-schedule with at most g gaps — for the modified instance where $p_k \leftarrow p$.

The following "expansion lemma" will be useful in the correctness proof. The proof of the lemma will appear in the final version.

Lemma 4. *Fix any k, s, g and p. Then*

(a) *If $U_{k,s,g}(p) < d_k$, then in the schedule realizing $U_{k,s,g}(p)$ the last block has at least one job other than k.*
(b) *If $p < p_k$ and $U_{k,s,g}(p) < d_k$, then $U_{k,s,g}(p+1) > U_{k,s,g}(p)$.*
(c) *If $p < p_k$ and $U_{k,s,g}(p) = d_k$ then $U_{k,s,g}(p+1) = d_k$ as well.*

We now define another table $P_{k,s,l,g}$. For any $k, s, l = 1, \ldots, n$, $g = 0, \ldots, n-1$, if $l = s$, then $P_{k,s,l,g} = 0$, otherwise

$$P_{k,s,l,g} = \min_p \{p + r_l - U_{k,s,g}(p)\},$$

where the minimum is taken over $0 \le p \le p_k$ such that l is the next job to be released after $U_{k,s,g}(p)$, that is $r_l = \min_{j<k} \{r_j : r_j > U_{k,s,g}(p)\}$. If there is no such p, we let $P_{k,s,l,g} = +\infty$. The intuition is that $P_{k,s,l,g}$ is the minimum amount of job k such that there is a (k,s)-schedule S with completion time r_l and at most g gaps. To be more precise, we also require that (for $P_{k,s,l,g} > 0$) S executes k at time $r_l - 1$ and that has maximal completion time among all schedules over the same set of jobs than S.

Our algorithm computes both tables $U_{k,s,g}$ and $P_{k,s,l,g}$. The intuition is this. Let S be a (k,s)-schedule with g gaps and maximum possible completion time u for the given values of k, s, g. Assume that S schedules job k and $u < d_k$. Moreover assume that k is scheduled in more than one interval, and let t be the end of the second last interval of k. Then S schedules at t some job $l < k$, for otherwise we could move some portion of k to the end, contradicting maximality of u. Furthermore, $r_l = t$ by the earliest deadline policy. Now the part of S up to r_l has some number of gaps, say h. The key idea is that the amount of job k in this part is minimal among all (k,s)-schedules with completion time r_k and at most h gaps, so this amount is equal to $P_{k,s,l,h}$.

Algorithm ALGB**.** For any $k = 0, ..., n$, $s = 1, ..., n$ and $g = 0, ..., n-1$, the algorithm computes $U_{k,s,g}$, and $P_{k,s,l,g}$ for all $l = 1, ..., n$. These values are computed in order of increasing values of k, with all $P_{k,s,l,g}$ computed before all $U_{k,s,g}$, using the following recurrence relations.

Computing $U_{k,s,g}$. For the base case $k = 0$ we let $U_{0,s,g} \leftarrow r_s$ for all s and g. For $k \geq 1$, $U_{k,s,g}$ is defined recursively as follows:

$$U_{k,s,g} \leftarrow \max_{l<k, h \leq g} \begin{cases} U_{k-1,s,g} & \text{if } r_k < r_s \text{ or } r_k \geq U_{k-1,s,g} \\ d_k & \text{if } P_{k,s,l,h} < p_k,\ \max_{j<k} r_j < U_{k-1,l,g-h-1} \\ & \text{and } d_k - U_{k-1,l,g-h-1} > p_k - P_{k,s,l,h} \\ d_k & \text{if } P_{k,s,l,h} < p_k,\ \max_{j<k} r_j < U_{k-1,l,g-h} \\ & \text{and } d_k - U_{k-1,l,g-h} \leq p_k - P_{k,s,l,h} \\ U_{k-1,l,g-h} + p_k - P_{k,s,l,h} & \text{if } P_{k,s,l,h} \leq p_k \text{ and} \\ & \nexists j < k : 0 \leq r_j - U_{k-1,l,g-h} < p_k - P_{k,s,l,g-h} \end{cases} \tag{3}$$

Computing $P_{k,s,l,g}$. If $r_s = r_l$, let $P_{k,s,s,g} \leftarrow 0$ for $r_k \leq r_s < d_k$ and $P_{k,s,s,g} = +\infty$ otherwise. Suppose now that $r_s < r_l$. If $r_k < r_s$ or $r_k \geq r_l$, let $P_{k,s,l,g} = +\infty$. For $r_s \leq r_k < r_l$, we compute $P_{k,s,l,g}$ recursively as follows:

$$P_{k,s,l,g} \leftarrow \min_{0 \leq h \leq k, j<k} \begin{cases} r_j - U_{k-1,s,h} + P_{k,j,l,g-h} & \text{if } r_k \leq U_{k-1,s,h},\ U_{k-1,s,h} < r_j \leq r_l, \text{ and} \\ & \nexists i < k : U_{k-1,s,h} \leq r_i < r_j \end{cases} \tag{4}$$

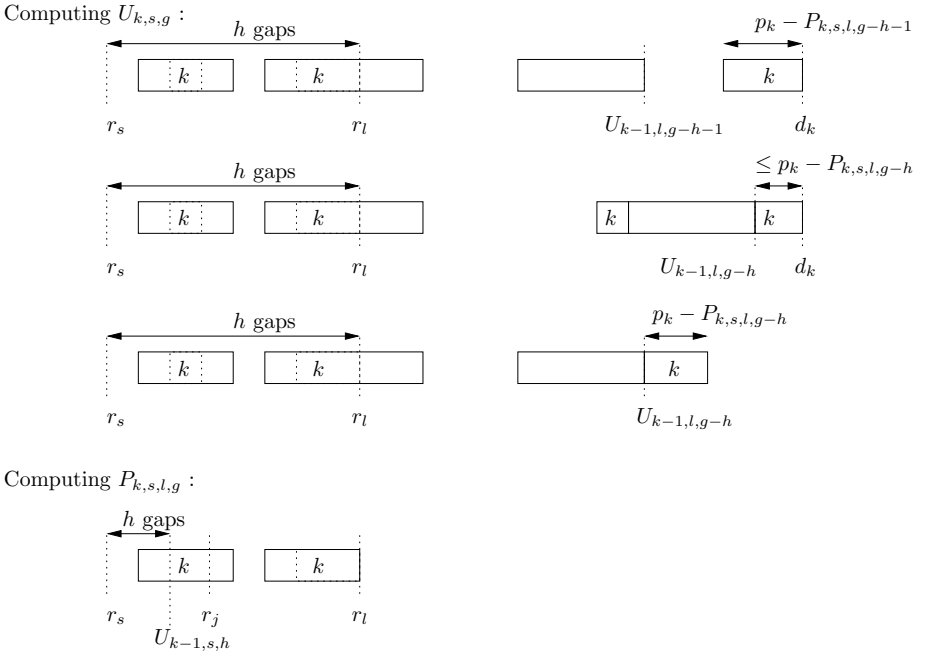


Fig. 2. Illustration of the cases in Algorithm ALGB

As usual, if the conditions in the minimum are not satisfied by any h, j, then $P_{k,s,l,g}$ is assumed to be $+\infty$. The cases considered in the algorithm are illustrated in Figure 2.

Theorem 2. *Algorithm* ALGB *correctly computes the optimum solution for* $1|r_j; \text{pmtn}; L = 1|E$*, and it can be implemented in time* $O(n^5)$*.*

The proof of this theorem will appear in the full version of the paper.

5 Minimizing the Energy

We now show how minimize the energy for an arbitrary L. This new algorithm consists of computing the table $U_{k,s,g}$ (using either Algorithm ALGA or ALGB) and an $O(n^2)$-time post-processing. Thus we can solve the problem for unit jobs in time $O(n^4)$ and for arbitrary-length jobs in time $O(n^5)$.

Recall that for this general cost model, the cost (energy) is defined as the sum over all gaps, of the minimum between L and the gap length. Call a gap *small* if its length is at most L. The idea of the algorithm is this: We show first that there is an optimal schedule where the short gaps divide the instance into disjoint sub-instances. For those sub-instances, the cost is simply the number of gaps times L. To compute the overall cost, we add to this quantity the total size of short gaps.

Given two schedules S, S' of the input instance, we say that S *dominates* S' if there is a time point t such that the supports of S and S' in the interval $(-\infty, t]$ are identical and at time t S schedules a job while S' is idle. This relation defines a total order on all schedules. The correctness of the algorithm relies on the following separation lemma.

Lemma 5. *There is an optimal schedule* S *with the following property: For any small gap* $[u, v]$ *of* S*, if a job* j *is scheduled at or after* v *then* $r_j \geq v$*.*

Proof. Among all optimal schedules, choose S to be the one not dominated by another optimal schedule, and let $[u, v]$ be a small gap in S. If there is a job j with $r_j < v$ and scheduled at some time unit $t \geq v$, then we can move this execution unit to the time unit $v - 1$. This will not increase the overall cost, since the cost in the small gap decreases by one, and the idle time unit created at t increases the cost at most by 1. The resulting schedule, however, dominates S – contradiction.

For any job s, define an *s-schedule* to be a (partial) schedule that schedules all jobs j with $r_j \geq r_s$. We use notation E_s to represent the minimum cost (energy) of an s-schedule, including the cost of the possible gap between r_s and its first block.

Algorithm ALGC**.** The algorithm first computes the table $U_{k,s,g}$, for all $k = 0, ..., n$, $s = 1, ..., n$, and $g = 0, 1, ..., n$, using either Algorithm ALGA or ALGB,

whichever applies. Then we use dynamic programming to compute all values E_s, in order of decreasing release times r_s:

$$E_s \leftarrow \min_{0 \le g \le n} \begin{cases} Lg & \text{if } U_{n,s,g} > \max_j r_j \\ Lg + r_l - u + E_l & \text{otherwise, where } u = U_{n,s,g}, r_l = \min\{r_j : r_j > u\} \end{cases} \quad (5)$$

The minimum energy of the whole instance is then E_1, where r_1 is the first release time. (Recall that the job 1 is assumed to be tight, so the schedule realizing E_1 will not have a gap at the beginning).

We now prove the correctness of Algorithm AlgC and analyze its running time.

Lemma 6. *For each job $s = 1, 2, ..., n$ there is an s-schedule S_s of cost at most E_s.*

Proof. The proof is by backward induction on r_s. In the base case, when s is the job with maximum release time, then we take S_s to be the schedule S_s that executes s at r_s. The cost of S_s is 0. Also, since $U_{n,s,0} > r_s$ we have $E_s = 0$, so the lemma holds.

Suppose now that for any $s' > s$ we have already constructed an s'-schedule $S_{s'}$ of cost at most $E_{s'}$. Let g be the value that realizes the minimum in (5).

If $U_{n,s,g} > \max_j r_j$ then, by Theorem 2, there is a schedule of all jobs released at or after r_s with at most g gaps. Let S_s be this schedule. Since each gap's cost is at most L, the total cost of S_s is at most Lg.

So now we can assume that $U_{n,s,g} \le \max_j r_j$. By the maximality of $U_{n,s,g}$, this inequality is strict. As in the algorithm, let l be the first job released after $U_{n,s,g}$. Choose a schedule S' realizing $U_{n,s,g}$. By induction, there exists an l-schedule S_l of cost at most E_l. We then define S_s as the disjoint union of S' and S_l. The cost of S' is at most Lg. Denote $u = U_{n,s,g}$. If $v \ge r_l$ is the first start time of a job in S_l, write E_l as $E_l = \max\{v - r_l, L\} + E'$. In other words, E' is the cost of the gaps in E_l excluding the gap before v (if any). Then the cost of S_s is at most $Lg + \max\{v - u, L\} + E' \le Lg + (r_l - u) + \max\{v - r_l, L\} + E' = Lg + r_l - u + E_l = E_s$.

Lemma 7. *For each job $s = 1, 2, ..., n$ there is an s-schedule S_s of cost at most E_s.*

Proof. For any job s, we now prove that any s-schedule Q has cost at least E_s. The proof is by backward induction on r_s. In the base case, when s is the job that is released last then $E_s = 0$, so the claim is true.

Suppose now that s is not the last job and let Q be an optimal s-schedule. By Lemma 5, we can assume that Q is not dominated by any other s-schedule with optimal cost. If Q does not have any small gaps then, denoting by g the number of gaps in Q, the cost of Q is $Lg \ge E_s$.

Otherwise, let $[u, v]$ be the first small gap in Q. Denote by Q' the segment of Q in $[r_s, u]$ and by Q'' the segment of Q in $[v, C_{\max}(S)]$. By Lemma 5, Q'' contains only jobs j with $r_j \ge v$. In particular the job l to be scheduled at v is released at $r_l = v$. By induction, the cost of Q'' is at least E_l.

Let g be the number of gaps in Q' and let R be the schedule realizing $U_{n,s,g}$. By the optimality of $U_{n,s,g}$, we have $C_{\max}(R) \geq u$. If $C_{\max}(R) = u$, then, by (5), the cost of Q is $Lg + r_l - u + E_l \geq E_s$, and we are done.

The remaining case is when $C_{\max}(R) > u$. By Lemma 1, this implies that there is a (n, s)-schedule R' with at most g gaps and $C_{\max}(R') \leq v$. But then we could replace Q' in Q by R', getting a schedule of cost strictly smaller than that of Q, contradicting the optimality of Q.

Theorem 3. *Algorithm* AlgC *correctly computes the optimum solution for* $1|r_j|E$*, and it can be implemented in time* $O(n^5)$*. Further, in the special case* $1|r_j; p_j = 1|E$*, it can be implemented in time* $O(n^4)$*.*

Proof. The correctness of Algorithm AlgC follows from Lemma 6 and Lemma 7, so it is sufficient to justify the time bound. By Theorem 1 and Theorem 2, we can compute the table $U_{k,s,g}$ in time $O(n^4)$ and $O(n^5)$ for unit jobs and arbitrary jobs, respectively. The post-processing, that is computing all values E_s, can be easily done in time $O(n^2 \log n)$, since we have n values E_s to compute, for each s we minimize over n values of g, and for fixed s and g we can find the index l in time $O(\log n)$ with binary search. (Finding this l can be in fact reduced to amortized time $O(1)$ if we process g in increasing order, for then the values of $U_{n,s,g}$, and thus also of l, increase monotonically as well).

6 Final Comments

We presented an $O(n^5)$-time algorithm for the minimum energy scheduling problem $1|r_j; \text{pmtn}|E$, and an $O(n^4)$ algorithm for $1|r_j; p_j = 1|E$.

Many open problems remain. Can the running times be improved further? In fact, fast — say, $O(n \log n)$-time — algorithms with low approximation ratios may be of interest as well.

To our knowledge, no work has been done on the multiprocessor case. Can our results be extended to more processors? Another generalization is to allow multiple power-down states [8,7]. Can this problem be solved in polynomial-time? In fact, the SS-PD problem discussed by Irani and Pruhs [8] is even more general as it involves speed scaling in addition to multiple power states, and its status remains open as well.

References

1. Augustine, J., Irani, S., Swamy, C.: Optimal power-down strategies. In: Proc. 45th Symp. Foundations of Computer Science (FOCS'04), pp. 530–539 (2004)
2. Baptiste, P.: Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In: Proc. 17th Annual ACM-SIAM symposium on Discrete Algorithms (SODA'06), pp. 364–367 (2006)
3. Chretienne, P.: On the no-wait single-machine scheduling problem. In: Proc. 7th Workshop on Models and Algorithms for Planning and Scheduling Problems (2005)

4. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H.Freeman and Co., New York (1979)
5. Irani, S., Gupta, R., Shukla, S.: Competitive analysis of dynamic power management strategies for systems with multiple power savings states. In: Proc. Conf. on Design, Automation and Test in Europe (DATE'02), p. 117 (2002)
6. Irani, S., Shukla, S., Gupta, R.: Algorithms for power savings. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03), pp. 37–46. ACM Press, New York (2003)
7. Irani, S., Shukla, S., Gupta, R.: Online strategies for dynamic power management in systems with multiple power-saving states. Trans. on Embedded Computing Sys. 2(3), 325–346 (2003)
8. Irani, S., Pruhs, K.R.: Algorithmic problems in power management. SIGACT News 36(2), 63–76 (2005)

k-Mismatch with Don't Cares

Raphaël Clifford[1], Klim Efremenko[2], Ely Porat[3], and Amir Rothschild[4]

[1] University of Bristol, Dept. of Computer Science, Bristol, BS8 1UB, UK
clifford@cs.bris.ac.uk
[2] Bar-Ilan University, Dept. of Computer Science, 52900 Ramat-Gan, Israel and Weizman institute, Dept. of Computer Science and Applied Mathematics, Rehovot, Israel
klimefrem@gmail.com
[3] Bar-Ilan University, Dept. of Computer Science, 52900 Ramat-Gan, Israel
porately@cs.biu.ac.il
[4] Tel-Aviv University, Dept. of computer science, Tel-Aviv, Israel and Bar-Ilan University, Dept. of Computer Science, 52900 Ramat-Gan, Israel
rotshch@post.tau.ac.il

Abstract. We give the first non-trivial algorithms for the k-mismatch pattern matching problem with don't cares. Given a text t of length n and a pattern p of length m with don't care symbols and a bound k, our algorithms find all the places that the pattern matches the text with at most k mismatches. We first give an $O(n(k + \log n \log\log n) \log m)$ time randomised solution which finds the correct answer with high probability. We then present a new deterministic $O(nk^2 \log^3 m)$ time solution that uses tools developed for group testing and finally an approach based on k-selectors that runs in $O(nk \operatorname{polylog} m)$ time but requires $O(\operatorname{poly} m)$ time preprocessing. In each case, the location of the mismatches at each alignment is also given at no extra cost.

1 Introduction

String matching, the problem of finding all the occurrences of a given pattern of length m in a text t of length n is a classic problem in computer science and can be solved in $O(n)$ time [4, 16]. The problem of determining the time complexity of exact matching with optional single character *don't care* symbols has also been well studied. Fischer and Paterson [13] presented the first solution based on fast Fourier transforms (FFT) with an $O(n \log m \log |\Sigma|)$ time algorithm in 1974[1], where Σ is the alphabet that the symbols are chosen from. Subsequently, the major challenge has been to remove this dependency on the alphabet size. Indyk [14] gave a randomised $O(n \log n)$ time algorithm which was followed by a simpler and slightly faster $O(n \log m)$ time randomised solution by Kalai [15]. In 2002, the first deterministic $O(n \log m)$ time solution was given [8] which was then further simplified in [7].

[1] Throughout this paper we assume the RAM model when giving the time complexity of the FFT. This is in order to be consistent with the large body of previous work on pattern matching with FFTs.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 151–162, 2007.

The key observation given by [7] but implicit in previous work is that for numeric strings, if there are no don't care symbols then for each location $1 \leq i \leq n - m + 1$ we can calculate

$$\sum_{j=1}^{m}(p_j - t_{i+j-1})^2 = \sum_{j=1}^{m}(p_j^2 - 2p_j t_{i+j-1} + t_{i+j-1}^2) \quad (1)$$

in $O(n \log m)$ time using FFTs. Wherever there is an exact match this sum will be exactly 0. If p and t are not numeric, then an arbitrary one-to-one mapping can be chosen from the alphabet to the set of positive integers $\mathbb{N}$. In the case of matching with don't cares, each don't care symbol in p or t is replaced by a 0 and the sum is modified to be

$$\sum_{j=1}^{m} p'_j t'_{i+j-1}(p_j - t_{i+j-1})^2$$

where $p'_j = 0$ ($t'_i = 0$) if p_j (t_i) is a don't care symbol and 1 otherwise. This sum equals 0 if and only if there is an exact match with don't cares and can also be computed in $O(n \log m)$ time using FFTs.

In many situations it is not enough to look for an exact match and some form of approximation is required. This might be to compensate for errors in a document, to look for similar images in a library or in the case of bioinformatics to look for functional similarities between genes or proteins for example.

In this paper we consider the widely used Hamming distance and in particular a bounded version of this problem which we call *k-mismatch don't cares*. Given a text t of length n and a pattern p of length m with don't care symbols and a bound k, our algorithms find all the places that the pattern matches the text with at most k mismatches. If the distance is greater than k, the algorithm need only report that fact and not give the actual Hamming distance.

2 Related Work and Previous Results

Much progress has been made in finding fast algorithms for the k-mismatch problem *without* don't cares over the last 20 years. $O(n\sqrt{m \log m})$ time solutions for the k-mismatch problem based on repeated applications of the FFT were given independently by both Abrahamson and Kosaraju in 1987 [1, 17]. Their algorithms are in fact independent of the bound k and report the Hamming distance at every position irrespective of its value. In 1985 Landau and Vishkin gave a beautiful $O(nk)$ algorithm that is not FFT based which uses constant time lowest common ancestor (LCA) operations on the suffix tree of p and t [18]. This was subsequently improved in [3] to $O(n\sqrt{k \log k})$ time by a method based on filtering and FFTs again. Approximations within a multiplicative factor of $(1+\epsilon)$ to the Hamming distance can also be found in $O(n/\epsilon^2 \log m)$ time [14]. A variant of the edit-distance problem (see e.g. [19]) called the k-difference problem with don't cares was considered in [2].

To the authors' knowledge, no non-naive algorithms have been given to date for the k-mismatch problem with don't cares. However, the $O(n\sqrt{m \log m})$ divide and conquer algorithm of Kosaraju and Abrahamson can be easily extended to handle don't cares in both the pattern and text with little extra work. This is because the algorithm counts matches and not mismatches. First we count the number of non don't care matches at each position i in $O(n\sqrt{m \log m})$ time. Then we need only subtract this number from the maximum possible number of non don't care matches in order to count the mismatches. To do this we create a new pattern string p' so that $p'_j = 1$ if p_j is not a don't care and $p_j = 0$ otherwise. A new text string t' is also made in the same way. The cross-correlation of p' and t' now gives us the maximum number of non don't care matches possible at each position. This single cross-correlation calculation takes $O(n \log m)$ time. Therefore the overall running time remains $O(n\sqrt{m \log m})$.

3 Our Results

We present the first non-naive solutions to the k-mismatch problem with don't cares. This problem does not appear to be amenable to any of the recent methods for solving the corresponding problem without don't cares. For example, the LCA based technique of Landau and Vishkin [18] requires the use of suffix trees to find longest common prefix matches between strings in constant time. It is not known how to solve this problem in even sublinear time when arbitrary numbers of don't cares are allowed. Similarly, it is not known how to apply filtering methods such as those in [3] when don't cares are permitted in both the pattern and the text.

We give two new algorithms that overcome these obstacles and provide substantial improvements to the known time complexities of the problem.

- In Section 5 we present a randomised algorithm that runs in $O(n(k + \log n \log \log n) \log m)$ time and gives the correct answer with high probability. The basic technique is to repeatedly sample subpatterns of p and find the positions of 1-mismatches in the text. In order to count the total number of mismatches overall we are also required at each stage to find the position and values of all the mismatches found so far. An further advantage of this approach is that our randomised algorithm can be made Las Vegas.
- In Section 6 we give a deterministic algorithm that runs in $O(nk^2 \log^3 m)$ time. The overall methodology is inspired by group testing but contains a number of innovations. Instead of performing a full group testing procedure we utilise the testing matrix to allow us to find 1-mismatches of selected subpatterns of p. Combining the results of these tests enables us deterministically to find the locations of the mismatches and to check that there are fewer than k.
- Finally we discuss the use of k-selectors instead of group testing and show that a deterministic $O(nk \operatorname{polylog} m)$ time solution can be found but that it requires preprocessing time polynomial in m.

4 Problem Definition and Preliminaries

Let Σ be a set of characters which we term the *alphabet*, and let ϕ be the don't care symbol. Let $t = t_1 t_2 \dots t_n \in \Sigma^n$ be the text and $p = p_1 p_2 \dots p_m \in \Sigma^m$ the pattern. Both the pattern and text may also include ϕ in their alphabet depending on the problem definition. The terms *symbol* and *character* are used interchangeably throughout. Similarly, we will sometimes refer to a *location* in a string and synonymously at other times a *position*.

- Define $HD(i)$ to be the Hamming distance between p and $t[i, \dots, i+m-1]$ and define the don't care symbol to match any symbol in the alphabet.
- Define $HD_k(i) = \begin{cases} HD(i) \text{ if } HD(i) \leq k \\ \perp \qquad\quad \text{otherwise} \end{cases}$
- We say that p is a k-mismatch at position i of t if $HD_k(i) \neq \perp$.

Our algorithms make extensive use of the fast Fourier transform (FFT). An important property of the FFT is that in the RAM model, the cross-correlation,

$$(t \otimes p)[i] \stackrel{\text{def}}{=} \sum_{j=1}^{m} p_j t_{i+j-1}, \quad 0 \leq i \leq n-m+1,$$

can be calculated accurately and efficiently in $O(n \log n)$ time (see e.g. [9], Chapter 32). By a standard trick of splitting the text into overlapping substrings of length $2m$, the running time can be further reduced to $O(n \log m)$. We will often assume that the text is of length $2m$ in the presentation of an algorithm or analysis in this paper and that the reader is familiar with this splitting technique.

5 Randomised k-Mismatch

We now present fast randomised algorithms that give the correct answer with high probability (w.h.p). Due to space constaints, some proof details have been omitted which will appear in the full version of the paper.

5.1 A Randomised Algorithm for the k-Mismatch Problem

Our randomised solution for the k-mismatch problem chooses a number of random subpatterns of p and performs a 1-mismatch on each one. Each 1-mismatch operation will tell us the location of a mismatch, if one occurs, at each position in the text. We will show that after $O(k \log n)$ iterations and w.h.p. all of the at most k mismatches at each location will have been identified and counted. As each 1-mismatch stage takes $O(n \log m)$ time the overall running time of the algorithm is $O(nk \log m \log n)$. We will then show how to reduce the running time further by recursively halving the number of sampling iterations.

1-Mismatch

The 1-mismatch problem is to determine if p matches $t[i, \ldots, i+m-1]$ with exactly one mismatch. Further, we identify the location of the mismatch for each such i. The method that we employ is to modify Equation 1 to give us the required information. The 1-MISMATCH algorithm is as follows:

1. Compute array $A_0[i] = \sum_j (p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$.
2. Compute array $A_1[i] = \sum_j (i+j-1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$.
3. If $A_0[i] \neq 0$ then $B[i] = A_1[i]/A_0[i]$. Otherwise let $B[i] = \perp$.
4. For each i s.t. $B[i] \neq \perp$, check to see if $(p[B[i] - i + 1] - t[B[i]])^2 = A_0[i]$. If this is not the case then let $B[i] = \perp$.

For any i where there are no mismatches between p and $t[i, \ldots, i+m-1]$, both $A_0[i] = 0$ and $A_1[i] = 0$. If there is exactly one mismatch then $B[i]$ is its location in t. Step 4 checks that the value of $B[i]$ came from no more than 1 mismatch. The following Theorem shows the overall time complexity.

Theorem 1. *The 1-mismatch problem can be solved in $O(n \log m)$ time.*

Proof. For a fixed location i in t, there are three cases for the 1-MISMATCH algorithm.

1. $HD(i) = 0 \Rightarrow A_0[i] = 0$ and $B[i]$ is correctly set to $\perp$.
2. $HD(i) = 1 \Rightarrow$ There is exactly one mismatch at some position I. Therefore $A_0[i] = (p_{I-i+1} - t_I)^2$ and $A_1[i] = I(p_{I-i+1} - t_I)^2$. Therefore $B[i] = A_1[i]/A_0[i] = I$ which gives the location of the mismatch in t.
3. $HD(i) > 1 \Rightarrow (p[B[i] - i + 1] - t[B[i]])^2 \neq A_0[i]$. Therefore $B[i]$ is correctly set to $\perp$.

The overall running time of the 1-mismatch algorithm is dominated by the time taken to perform the FFTs and is therefore $O(n \log m)$. □

Sampling and Matching

We show how the 1-MISMATCH algorithm can be repeatedly applied to random subpatterns of p to solve the full k-mismatch problem. The 1-MISMATCH algorithm allows us to determine if a pattern has a 1-mismatch with any position in the text and also gives us the position of each mismatch. First a subpattern p^* of p is selected and then used to find 1-mismatches in the text. We choose the subpattern at random by selecting m/k locations j uniformly at random from the pattern. The subpattern is set so that $p^*_j = p_j$ for those chosen locations. We set the characters of all other positions in p^* to be the don't care symbol and ensure $|p^*| = |p|$. We then run the 1-mismatch algorithm using p^* and t. This whole "sample and match" process is repeated $O(k \log n)$ times, each time keeping record of where any single mismatch occurs for each index i in the text. Algorithm 1 sets out the main steps.

Input: Pattern p, text t and an integer k
Output: $O[i] = HD_k(p, t[i, \ldots, i+m-1])$
Sample $O(k \log n)$ subpatterns;
for *subpattern* p^* **do**
  1-MISMATCH(p^*, t);
end
Let $L[i]$ = total number distinct mismatches found at location i in t;
$O[i] = L[i]$ if $L[i] \leq k$, otherwise $L[i] = \perp$;

Algorithm 1. Randomised k-mismatch with don't cares

We analyse the algorithm for a single location in the text and show that with w.h.p. the Hamming distance is correctly computed if there are no more than k mismatches. This probabilistic bound is then sufficient to show that the Hamming distance will be correctly computed for all locations i in t w.h.p. The proof of the following Lemma is an application of the *coupon collector's problem* (see e.g. [12]) and is omitted for space reasons.

Lemma 1. *For a single given location i in the text such that $HD(i) \leq k$, $HD(i)$ will be correctly computed with probability at least $1 - O(e^{-c/e})$ after $3k \log k + ck$ iterations of the sample and match stage, where $c > 1$.*

The running time of the full algorithm follows immediately from the fact that we need to ensure our bound holds not only for one position but for every one in the text simultaneouly. Therefore c must be set to be $O(\log n)$ making a total of $O(k \log k + k \log n)$ iterations, each of which takes $O(n \log m)$ time. The remaining element is how to determine if the Hamming distance is greater than k. Notice that 1-MISMATCH gives us not only the location I of the error but also the value $A_0[I - i + 1] = (p_{I-i+1} - t_I)^2$ for each i where there is a 1-mismatch. If we first compute $C = p \otimes t$ then we can "correct" this cross-correlation for each distinct 1-mismatch found by subtracting $A_0[i]$ from $C[i]$. In this way, we can check whether we have found all the mismatches at a given position in $O(k)$ time. If we have found up to k different mismatches at some position i and after performing all the k corrections $C[i] \neq 0$, then there is no k-mismatch at that position.

Theorem 2. *The k-mismatch problem with don't cares can be solved w.h.p. in $O(nk \log m \log n))$ time.*

Proof. By Lemma 1, $3k \log k + c'k \log n$ sample and match iterations are sufficient w.h.p. to compute $HD(i)$ for every location i where $HD(i) \leq k$. After $O(k \log n)$ iterations we can check if there is a k-mismatch at position i by subtracting the contributions of the mismatches from the value of $(p \otimes t)[i]$ and checking if the result is zero. If after $O(k \log n)$ samples the corrected value at position i is not zero then we know w.h.p. that $HD(i) > k$. Each call to 1-mismatch takes $O(n \log m)$ time. Therefore the total running time is $O(nk \log m \log n)$ as $k \leq m \leq n$. □

An $O(n(k + \log n \log\log n) \log m)$ Time Recursive Algorithm

We discuss here how to improve the time complexity of the previous algorithm from $O(nk \log m \log n)$ to $O(n(k + \log n \log\log n) \log m)$. The approach is recursive and requires us to halve the number of mismatches we are looking for at each turn. To simplify the explanation consider the worst case, where the number of mismatches at a location is at least k. If the number of mismatches is less than k then our algorithm will find the mismatches more quickly and so the bounds given still hold.

Lemma 2 shows that we can find $k/2$ of the required mismatches in $O(k)$ time. We modify the 1-MISMATCH algorithm to allow us at each iteration to correct the arrays A_0 and A_1 using the information from the distinct mismatches found so far. This is performed before attempting to determine if there is a new 1-mismatch. As before we will be given not only the location of each error but the contributions they make to the overall cross-correlation.

Lemma 2. *For a given position i in the text after ck iterations of* 1-MISMATCH*, where $c > e$, the locations and values of $k/2$ different mismatches will be found with probability at least $1 - e^{-\Theta(ck)}$.*

Proof. We assume for simplicity the worst case where the number of mismatches remaining to be found is k. The probability of finding a 1-MISMATCH after one iteration is $\geq e^{-1}$. Therefore the probability that fewer than $ck/2e$ not necessarily distinct mismatches are found after ck iterations $\leq e^{-\Theta(ck)}$. On the other hand, given $c'k$ not necessarily distinct mismatches, the probability that fewer than $k/2$ are distinct $\leq ((k/2)^{c'k}\binom{k}{k/2}/k^{c'k})$. Therefore, assuming we have $c > e$ and ck iterations of 1-MISMATCH, the probability of finding fewer than $k/2$ mismatches is at most $e^{-\Theta(ck)}$ and the result is proved.

The recursive k-mismatch algorithm is as follows. First we fix some constant $c > 1$.

1. Sample $\max(ck, c\log(n))$ subpatterns. Each has the symbols at m/k positions selected at random copied from p
2. For each subpattern p^*, compute A_0 and A_1 as in 1-MISMATCH
3. Correct A_0 and A_1 (resulting from step 2) using the information from the distinct mismatches discovered so far
4. Locate up to $k/2$ different new mismatches at each position in the text and their associated error locations and the contributions they make to A_0 and A_1
5. Set $k = k/2$ and repeat from step 1

By Lemma 2 we can find $k/2$ different mismatches w.h.p. after having sampled ck different subpatterns. Further, we can find not only the locations of all these mismatches at each position in the text but also how much the mismatches contributed to the cross-correlation calculations. At the next iteration of the algorithm we subtract all previously found contributions from the mismatches found so far. In this way we are able to correct the values found by

1-MISMATCH. However, in order to find the last remaining mismatches w.h.p. we must perform some more work. If $k < c \log n$ we carry on doubling the sample rate for the pattern but maintain the number of subpatterns sampled at $c \log n$. Combined with other minor modifications, the following theorem gives the final time complexity of this modified recursive algorithm.

Theorem 3. *The k-mismatch with don't cares problem can be solved w.h.p in* $O(n(k + \log n \log \log n) \log m)$ *time.*

Proof. Each level of the recursion where $k \geq c \log n$ gives the correct answer w.h.p. by Lemma 2. As there are only $\log k$ of these levels the bound also holds for all levels simultaneously w.h.p. For the last $O(\log \log n)$ steps when $k < c \log n$, we fix the number of subpatterns sampled to be $c \log n$ in order to ensure that the bound holds for all $O(n)$ positions in the text w.h.p.

6 Deterministic *k*-Mismatch with Don't Cares

In this section we give an $O(nk^2 \log^3 m)$ time deterministic solution for the k-mismatch with don't cares problem. Our algorithm uses a testing matrix designed for group testing and combines it with the 1-MISMATCH algorithm from Section 5.1.

The group testing problem can be described as follows. Consider a set of n items, each of which can be *defective* or *non-defective*. The task is to identify the defective items using the minimum number of tests. Each test works on a group of items simultaneously and returns whether that group contains at least one defective item or not. If a defective item is present in a group then the test result is said to be positive, otherwise it is negative.

Group testing has a long history dating back to at least 1943 [10]. In this early work the problem of detecting syphilitic men for induction into the United States military using the minimum number of laboratory tests was considered. Subsequently a large literature has built up around the subject and we refer the interested reader to [11] for a comprehensive survey of the topic. Our use of group testing to find mismatches considers the squared L_2 norm as before and equates a positive value (i.e. $(p_j - t_{i+j-1})^2 > 0$ for some i and j) with a defective item.

Some basic definitions are required first (see [11] for a full discussion of the definitions and the context in which they are used). A $0-1$ matrix sets out a sequence of tests in the following way. A 1 at position (i, j) specifies that in test i, the jth element should be included. A 0 specifies that the jth element should be excluded from that test. To detect which items were defective after the tests have completed we require that the result of the tests is unique as long as the number of defective items is less than or equal to k. This condition is expressed more formally as follows.

Definition 1. *A* $t \times n$ $0-1$ *matrix is* k^-*-separable if the bitwise disjunction of each set of at most* k *columns is distinct.*

A k^--separable matrix will uniquely identify the defective items, however, in our algorithm we require that the matrix will obey a stronger condition:

Definition 2. *A $t \times n$ $0-1$ matrix is k^--disjunct if no union of up to k columns* covers *any other column. A vector a is said to* cover *b, if their bitwise disjunction equals a (i.e. $a \vee b = a$). Clearly, a k^--disjunct matrix is k^--separable.*

The first stage of our algorithm will be to create a k^--disjunct matrix with m columns and the minimum number of rows as possible. The result is a k^- disjunct matrix with m columns and $O(k^2 \log^2 m)$ rows (see e.g. [11]). There is a lower bound of $\Omega(k^2 \log_k m)$ for the number of rows (tests) in a k^--disjunct matrix, however explicit constructions are not currently known [6].

Although it is possible to find all defective items from the result of the group tests, in our case we would have to perform this work for all $O(m)$ positions in the text. Therefore we employ a different method to find the mismatches. The following Lemma allows us to use a k^--disjunct matrix M to find 1-mismatches of subpatterns of p.

Lemma 3. *If there are at most k defective items then every defective item will occur at least once on its own in one of the tests in the k^--disjunct matrix M.*

Proof. The proof is by contradiction. Consider a defective item i. Assume that for every test in M, i is included with some other defective item. This means that ith column in M is covered by the union of the columns of the rest of the defective items. There are at most $k-1$ such columns which contradicts the assumption that M is k^--disjunct. □

The Deterministic Algorithm

We can now give a full deterministic algorithm for the k-mismatch problem with don't cares. We repeatedly run 1-MISMATCH on subpatterns of p as before. However, this time we use the testing matrix M to only calculate the sum for a subset of indices in the pattern. We therefore redefine A_0 and A_1 from 1-MISMATCH as follows.

$$A_0[i] = \sum (p_j - t_{i+j-1})^2 M_{\ell,j} p'_j t'_{i+j-1}$$

and

$$A_1[i] = \sum_j (i+j-1)(p_j - t_{i+j-1})^2 M_{\ell,j} p'_j t'_{i+j-1}$$

where p'_j (t'_i) equals 0 if p_j (t_i) is a don't care symbol and 1 otherwise.

The above method also gives us the position of the mismatch, and we can therefore check the mismatches in those positions in constant time per mismatch.

To finish the algorithm, we check whether all the mismatches have been found at each position in the text. This can be done in the same way as in Section 5 by "correcting" the cross-correlation calculations. First, we compute the sum:

$$D[i] = \sum (p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$$

Input: Pattern p, text t and an integer k
Output: $O[i] = HD_k(p, t[i, \ldots, i+m-1])$
$M \leftarrow O(k^2 \log^2 m) \times m$, k^--disjunct matrix;
for $1 \leq \ell \leq rows(M)$ **do**
  $S_\ell \leftarrow 1-\text{mismatch}(M_\ell, t)$;
end
$L[i] \leftarrow$ number of distinct 1-mismatches at position i in t;
Check at each position i in t that all mismatches were found;
$O[i] = L[i]$, if all mismatches found, otherwise $L[i] = \perp$;

Algorithm 2. Deterministic k-mismatch with don't cares

Then in turn we subtract from $D[i]$ the contributions for every 1-mismatch found. This leaves us with 0 if and only if all the mismatches were found. Otherwise, we know that there were more than k mismatches can report that fact. The time for this stage is again $O(m \log m + mk)$ and so does not affect the overall running time.

Theorem 4. *The k-mismatch with don't cares problem can be solved in $O(nk^2 \log^3 m)$ time.*

Proof. Algorithm 2 sets out the main steps of the algorithm. The k^--disjunct matrix can be constructed in $O(mk^2 \log^2 m)$ time. For each ℓ, the 1-mismatch calculation to compute S_ℓ takes $O(m \log m)$ time. Up to k distinct 1-mismatches can be found and counted in $O(mk)$ further time. Finally, we can check and correct all mismatches found in $O(m \log m + mk)$ time. The total running time is therefore $O(nk^2 \log^3 m)$ as required. □

Further Deterministic Speedups

k-selectors are generalizations of group testing which can also be used to solve our problem in a similar way. With little change k-selectors could be used to replace the k^--disjunct matrices for example. This is of interest as the lower bound for the number of tests for a selector is $\Omega(k \log_k m)$ whereas it is $\Omega(k^2 \log_k m)$ for group testing [6]. Using selectors instead of k^--disjunct matrices might save us a factor of k in the running time. Unfortunately, there are no known efficient algorithms for building selectors of size smaller then $O(k \operatorname{polylog} m)$, and those algorithms take $O(\operatorname{poly} m)$ time [5]. So while we save a factor of k, we lose a factor of $\operatorname{polylog} m$, and $\operatorname{poly} m$ preprocess time. We finish with the following two remarks:

Remark 1. The k-mismatch with don't cares problem can be solved using k-selectors instead of k^- disjunct matrices with the cost of an $O(\log m)$ factor in the algorithm. The extra $O(\log m)$ factor is due to the fact that one scan for mismatches is no longer sufficient.

Remark 2. As the best construction of k-selectors is of size $O(k \operatorname{polylog} m)$ and takes $O(\operatorname{poly} m)$ time to construct, the new algorithm will then run in

$O(nk \operatorname{polylog} m)$ time but with $O(\operatorname{poly} m)$ preprocessing time. However, more efficient constructions of selectors will translate into more efficient algorithms for the k-mismatch problem.

7 Conclusion

We have presented the first non-trivial algorithms for the k-mismatch with don't cares problem. We conjecture that the gap between their deterministic and randomised complexities can be closed. A further interesting open question is whether an $\tilde{O}(n\sqrt{k})$ algorithm can be found to match the fastest known solution for the problem without don't care symbols.

References

[1] Abrahamson, K.: Generalized string matching. SIAM journal on Computing 16(6), 1039–1051 (1987)
[2] Akutsu, T.: Approximate string matching with don't care characters. Information Procesing Letters 55, 235–239 (1995)
[3] Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. J. Algorithms 50(2), 257–275 (2004)
[4] Boyer, R.S., Moore, J.S.: A fast string matching algorithm. Communications of the ACM 20, 762–772 (1977)
[5] Chlebus, B.S., Kowalski, D.R.: Almost optimal explicit selectors. In: Liśkiewicz, M., Reischuk, R. (eds.) FCT 2005. LNCS, vol. 3623, Springer, Heidelberg (2005)
[6] Clementi, A.E.F., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (SODA '01), pp. 709–718. ACM Press, New York (2001)
[7] Clifford, P., Clifford, R.: Simple deterministic wildcard matching. Information Processing Letters 101(2), 53–54 (2007)
[8] Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proceedings of the Annual ACM Symposium on Theory of Computing, pp. 592–601. ACM Press, New York (2002)
[9] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
[10] Dorfman, R.: The detection of defective members of large populations. The Annals of Mathematical Statistics 14(4), 436–440 (1943)
[11] Du, D.Z., Hwang, F.K.: Combinatorial Group Testing and its Applications, 2nd edn. Series on Applied Mathematics, vol. 12. World Scientific, Singapore (2000)
[12] Feller, W.: An introduction to probability theory and its applications, vol. 1. Wiley, Chichester (1968)
[13] Fischer, M., Paterson, M.: String matching and other products. In: Karp, R. (ed.) Proceedings of the 7th SIAM-AMS Complexity of Computation, pp. 113–125 (1974)
[14] Indyk, P.: Faster algorithms for string matching problems: Matching the convolution bound. In: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, pp. 166–173 (1998)

[15] Kalai, A.: Efficient pattern-matching with don't cares. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 655–656, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2002)
[16] Knuth, D.E., Morris, J.H., Pratt, V.B.: Fast pattern matching in strings. SIAM Journal of Computing 6, 323–350 (1977)
[17] Kosaraju, S.R.: Efficient string matching. Manuscript (1987)
[18] Landau, G.M., Vishkin, U.: Efficient string matching with k mismatches. Theoretical Computer Science 43, 239–249 (1986)
[19] Landau, G.M., Vishkin, U.: Efficient string matching in the presence of errors. In: Proc. 26th IEEE FOCS, pp. 126–126. IEEE Computer Society Press, Los Alamitos (1985)

Finding Branch-Decompositions and Rank-Decompositions

Petr Hliněný[1,⋆] and Sang-il Oum[2,⋆⋆]

[1] Faculty of Informatics, Masaryk University,
Botanická 68a, 602 00 Brno, Czech Republic
hlineny@fi.muni.cz
[2] Department of Combinatorics and Optimization,
University of Waterloo, Waterloo, Ontario N2L 3G1 Canada
sangil@math.uwaterloo.ca

Abstract. We present a new algorithm that can output the rank-decomposition of width at most k of a graph if such exists. For that we use an algorithm that, for an input matroid represented over a fixed finite field, outputs its branch-decomposition of width at most k if such exists. This algorithm works also for partitioned matroids. Both these algorithms are fixed-parameter tractable, that is, they run in time $O(n^3)$ for each fixed value of k where n is the number of vertices / elements of the input. (The previous best algorithm for construction of a branch-decomposition or a rank-decomposition of optimal width due to Oum and Seymour [Testing branch-width. J. Combin. Theory Ser. B, **97**(3) (2007) 385–393] is not fixed-parameter tractable).

Keywords: Rank-width, clique-width, branch-width, fixed parameter tractable algorithm, graph, matroid.

1 Introduction

Many graph problems are known to be NP-hard in general. But for practical application we still need to solve them. One method to solve them is to restrict the input graph to have a certain structure. Clique-width, defined by Courcelle and Olariu [4], is very useful for that purpose. Many hard graph problems (in particular all those expressible in MSO logic of adjacency graphs) are solvable in polynomial time as long as the input graph has bounded clique-width and is given in the form of the decomposition for clique-width, called a *k-expression* [3,24,6,14,10]. A k-expression is an algebraic expression with the following four operations on vertex-labeled graph with k labels: create a new vertex with label i; take the disjoint union of two labeled graphs; add all edges between vertices of label i and label j; and relabel all vertices with label i to have label j. However, for fixed $k > 3$, it is not known how to find a k-expression of an input graph having clique-width at most k. (If $k \leq 3$, then it has been shown in [2,1]).

⋆ Supported by research intent MSM0021622419 of the Czech Ministry of Education.
⋆⋆ Partially supported by NSF grant 0354742.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 163–174, 2007.

Rank-width is another graph complexity measure introduced by Oum and Seymour [19], aiming at construction of an $f(k)$-expression of the input graph having clique-width k for some fixed function f in polynomial time. Rank-width is defined (Section 6) as the branch-width (see Section 2) of the *cut-rank* function of graphs. Rank-width turns out to be very useful for algorithms on graphs of bounded clique-width, since a class of graphs has bounded rank-width if and only if it has bounded clique-width. In fact, if rank-width of a graph is k, then its clique-width lies between k and $2^{k+1}-1$ [19] and an expression can be constructed from a rank-decomposition of width k.

In this paper, we are mainly interested in the following problem:

- Find a fixed-parameter tractable algorithm that outputs a rank-decomposition of width at most k if the rank-width of an input graph (with more than one vertex) is at most k.

The first rank-width algorithm by Oum and Seymour [19] only finds a rank-decomposition of width at most $3k+1$ for n-vertex graphs of rank-width at most k in time $O(n^9 \log n)$. This algorithm has been improved by Oum [18] to output a rank-decomposition of width at most $3k$ in time $O(n^3)$. Using this approximation algorithm and finiteness of excluded vertex-minors [17], Courcelle and Oum [5] have constructed an $O(n^3)$-time algorithm to decide whether a graph has rank-width at most k. However, this is only a decision algorithm; if the rank-width is at most k, then this algorithm verifies that the input graph contains none of the excluded graphs for rank-width at most k as a vertex-minor. It does *not* output a rank-decomposition showing that the graph indeed has rank-width at most k.

In another paper, Oum and Seymour [20] have constructed a polynomial-time algorithm that can output a rank-decomposition of width at most k for graphs of rank-width at most k. However, it is not fixed-parameter tractable; its running time is $O(n^{8k+12} \log n)$. Obviously, it is very desirable to have a fixed-parameter tractable algorithm to output such an "optimal" rank-decomposition, because most algorithms on graphs of bounded clique-width require a k-expression on their input. So far probably the only known efficient way of constructing an expression with bounded number of labels for a given graph of bounded clique-width uses rank-decompositions.

In this paper, we present an affirmative answer to the above problem. An amusing aspect of our solution is that we deeply use submodular functions and matroids to solve the rank-decomposition problem, which shows (somehow unexpectedly) a "truly geometrical" nature of this graph-theoretical problem. In fact we solve the following related problem on matroids, too.

- Find a fixed-parameter tractable algorithm that, given a matroid represented by a matrix over a fixed finite field, outputs a branch-decomposition of width at most k if the branch-width of the input matroid is at most k.

Here we actually bring together two separate lines of research; Oum and Seymour's above sketched work on rank-width and on branch-width of submodular functions, with Hliněný's work [12,13] on parametrized algorithms for matroids over finite fields, to give the final solution of our first problem — Theorem 6.3.

We lastly remark that the following (indeed widely expected) hardness result has been given only recently by Fellows, Rosamond, Rotics, and Szeider [7]; it is NP-hard to find graph clique-width. To argue that it is NP-hard to find rank-width, we combine some known results: Hicks and McMurray Jr. [11] (independently Mazoit and Thomassé [15]) recently proved that the branch-width of the cycle matroid of a graph is equal to the branch-width of the graph if it is 2-connected. Hence we can reduce (Section 6) the problem of finding branch-width of a graph to finding rank-width of a certain bipartite graph, and finding graph branch-width is NP-hard as shown by Seymour and Thomas [23].

Our paper is structured as follows: The next section briefly introduces definitions of branch-width, partitions, matroids and the amalgam operation on matroids. After that (Section 3) we explain the notion of so-called titanic partitions, which we further use to "model" partitioned matroids in ordinary matroids. (At this point it is worth to note that partitioned matroids present the key tool that allows us to shift from a branch-width-testing algorithm [12] to a construction of an "optimal" branch-decomposition, see Theorem 4.4, and of a rank-decomposition.) In Section 4, we will discuss a simple but slow algorithm for matroid branch-decompositions. In Section 5, we will present a faster algorithm. As the main application we then use our result to give an algorithm for constructing a rank-decomposition of optimal width of a graph in Section 6.

2 Definitions

Branch-Width. Let $\mathbb{Z}$ be the set of integers. For a finite set V, a function $f : 2^V \to \mathbb{Z}$ is called *symmetric* if $f(X) = f(V \setminus X)$ for all $X \subseteq V$, and is called *submodular* if $f(X) + f(Y) \geq f(X \cap Y) + f(X \cup Y)$ for all subsets X, Y of V. A tree is *subcubic* if all vertices have degree 1 or 3. For a symmetric submodular function $f : 2^V \to \mathbb{Z}$ on a finite set V, the branch-width is defined as follows.

A *branch-decomposition* of the symmetric submodular function f is a pair (T, μ) of a subcubic tree T and a bijective function $\mu : V \to \{t : t \text{ is a leaf of } T\}$. (If $|V| \leq 1$ then f admits no branch-decomposition.) For an edge e of T, the connected components of $T \setminus e$ induce a partition (X, Y) of the set of leaves of T. (In such a case, we say that $\mu^{-1}(X)$ (or $\mu^{-1}(Y)$) is *displayed* by e in the branch-decomposition (T, μ). We also say that V and $\emptyset$ are displayed by the branch-decomposition.) The *width* of an edge e of a branch-decomposition (T, μ) is $f(\mu^{-1}(X))$. The *width* of (T, μ) is the maximum width of all edges of T. The *branch-width* of f, denoted by $\mathrm{bw}(f)$, is the minimum of the width of all branch-decompositions of f. (If $|V| \leq 1$, we define $\mathrm{bw}(f) = f(\emptyset)$.)

A natural application of this definition is the branch-width of a graph, as introduced by Robertson and Seymour [22] along with better known tree-width, and its direct matroidal counterpart below in this section. We also refer to further formal definition of rank-width in Section 6.

Partitions. A *partition* $\mathcal{P}$ of V is a collection of nonempty pairwise disjoint subsets of V whose union is equal to V. Each element of $\mathcal{P}$ is called a *part*. For

a symmetric submodular function f on 2^V and a partition $\mathcal{P}$ of V, let $f^{\mathcal{P}}$ be a function on $2^{\mathcal{P}}$ (also symmetric and submodular) such that $f^{\mathcal{P}}(X) = f(\cup_{Y\in X} Y)$. The *width* of a partition $\mathcal{P}$ is $f(\mathcal{P}) = \max\{f(Y) : Y \in \mathcal{P}\}$.

Matroids. We refer to Oxley [21] in our matroid terminology. A *matroid* is a pair $M = (E, \mathcal{B})$ where $E = E(M)$ is the ground set of M (elements of M), and $\mathcal{B} \subseteq 2^E$ is a nonempty collection of *bases* of M, no two of which are in an inclusion. Moreover, matroid bases satisfy the "exchange axiom": if $B_1, B_2 \in \mathcal{B}$ and $x \in B_1 \setminus B_2$, then there is $y \in B_2 \setminus B_1$ such that $(B_1 \setminus \{x\}) \cup \{y\} \in \mathcal{B}$. A typical example of a matroid is given by a set of vectors (forming the columns of a matrix $\boldsymbol{A}$) with usual linear independence. The matrix $\boldsymbol{A}$ is then called a *representation* of the matroid.

All matroid bases have the same cardinality called the *rank* $r(M)$ of the matroid. Subsets of bases are called *independent sets*, and the remaining sets are *dependent*. A matroid M is *uniform* if all subsets of $E(M)$ of size $r(M)$ are the bases, and M is *free* if $E(M)$ is a basis. The *rank function* $r_M(X)$ in M is the maximum cardinality of an independent subset of a set $X \subseteq E(M)$. The *dual matroid* M^* is defined on the same ground set with the bases as set-complements of the bases of M. For a subset X of E, the *deletion* $M \setminus X$ of X from M or the *restriction* $M \restriction (E \setminus X)$ of M to $E \setminus X$, is the matroid on $E \setminus X$ in which $Y \subseteq E \setminus X$ is independent in $M \setminus X$ if and only if Y is an independent set of M. The *contraction* M/X of X in M is the matroid $(M^* \setminus X)^*$. Matroids of the form $M/X \setminus Y$ are called *minors* of M.

To define the branch-width of a matroid, we consider its (symmetric and submodular) connectivity function $\lambda_M(X) = r_M(X) + r_M(E \setminus X) - r_M(E) + 1$ defined for all subsets $X \subseteq E = E(M)$. A "*geometric*" meaning is that the subspaces spanned by X and $E \setminus X$ intersect in a subspace of rank $\lambda_M(X) - 1$. *Branch-width* $\mathrm{bw}(M)$ and *branch-decompositions* of a matroid M are defined as the branch-width and branch-decompositions of λ_M. A pair $(M, \mathcal{P})$ is called a *partitioned matroid* if M is a matroid and $\mathcal{P}$ is a partition of $E(M)$. A *connectivity function* of a partitioned matroid $(M, \mathcal{P})$ is defined as $\lambda^{\mathcal{P}}_M$. *Branch-width* $\mathrm{bw}(M, \mathcal{P})$ and *branch-decompositions* of a partitioned matroid $(M, \mathcal{P})$ are defined as branch-width, branch-decompositions of $\lambda^{\mathcal{P}}_M$.

Amalgams of Matroids. Let M_1, M_2 be matroids on E_1, E_2 respectively and $T = E_1 \cap E_2$. Moreover let us assume that $M_1 \restriction T = M_2 \restriction T$. If M is a matroid on

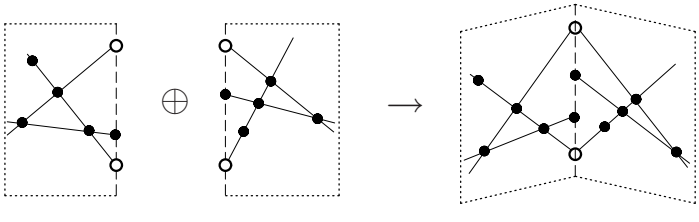


Fig. 1. A "geometrical" illustration of an amalgam of two matroids, in which hollow points are the shared elements T

$E_1 \cup E_2$ such that $M \restriction E_1 = M_1$ and $M \restriction E_2 = M_2$, then M is called an *amalgam* of M_1 and M_2 (see Fig. 1). It is known that an amalgam of two matroids need not exist (and need not be unique). However, in a special case we shall use here (see also Proposition 4.2), its existence easily follows from [21, (12.4.2)];

Lemma 2.1. *If $M_1 \restriction T$ is free, then an amalgam of M_1 and M_2 exists.*

3 Titanic Partitions and Gadgets

Let V be a finite set and f be a symmetric submodular function on 2^V. A subset X of V is called *titanic* with respect to f if whenever A_1, A_2, A_3 are pairwise disjoint subsets of X such that $A_1 \cup A_2 \cup A_3 = X$, there is $i \in \{1, 2, 3\}$ such that $f(A_i) \geq f(X)$. A partition $\mathcal{P}$ of V is called *titanic* with respect to f if every part of $\mathcal{P}$ is titanic with respect to f. The following lemma is equivalent to a lemma by Geelen, Gerards, and Whittle [9, 4.4], which generalizes a result of Robertson and Seymour [22, (8.3)].

Lemma 3.1. *Let V be a finite set and f be a symmetric submodular function on 2^V of branch-width at most k. If $\mathcal{P}$ is a titanic partition of width at most k with respect to f, then the branch-width of $f^{\mathcal{P}}$ is at most k.*

The purpose of this section is to show how a partitioned matroid may be "modeled" by an ordinary matroid having the same branch-width. We aim to transform a partitioned matroid $(M, \mathcal{P})$ to another partitioned matroid $(M^{\#}, \mathcal{P}^{\#})$, such that they have the same branch-width and $\mathcal{P}^{\#}$ is a titanic partition with respect to $\lambda_{M^{\#}}$.

We may assume each part T of $\mathcal{P}$ satisfies $\lambda_M(T) = |T| + 1$ if $|T| > 1$, because otherwise we can contract or delete elements in T while preserving $\mathrm{bw}(M, \mathcal{P})$. This means that $M \restriction T$ is a free matroid. For each part T of $(M, \mathcal{P})$, if $|T| > 1$, then we define a matroid U_T as a rank-$|T|$ uniform matroid on the ground set $E_T = E(U_T)$ such that $|E_T| = 3|T| - 2$, $E(M) \cap E_T = T$, and $E_T \cap E_{T'} = \emptyset$ if $T' \neq T$ is a part of $\mathcal{P}$ and $|T'| > 1$. Since $M \restriction T = U_T \restriction T$ is a free matroid, an amalgam of M and U_T exists by Lemma 2.1. The following theorem is based on a lemma stating that set $E(U_T)$ is always titanic in this amalgam.

Theorem 3.2. *Let $(M_0, \mathcal{P}_0)$ be a partitioned matroid and let $T_1, T_2, \ldots, T_m$ be the parts of $\mathcal{P}_0$ having at least two elements. Assume that $\lambda_{M_0}(T_i) = |T_i| + 1$ for every $i \in \{1, 2, \ldots, m\}$. For all $i = 1, 2, \ldots, m$, let M_i be an amalgam of M_{i-1} and U_{T_i}. Then the branch-width of M_m is equal to the branch-width of the partitioned matroid $(M_0, \mathcal{P}_0)$.*

We call resulting $M^{\#} = M_m$ the *normalized matroid* of $(M_0, \mathcal{P}_0)$.

4 Branch-Decompositions of Represented Partitioned Matroids

We now specialize the above ideas to the case of representable matroids. We aim to provide an efficient algorithm for testing small branch-width on such

matroids. For the rest of our paper, a *represented matroid* is the vector matroid of a (given) matrix over a *fixed* finite field. We also write $\mathbb{F}$-*represented* matroid to explicitly refer to the field $\mathbb{F}$. In other words, an $\mathbb{F}$-represented matroid is a set of points (a *point configuration*) in a (finite) projective geometry over $\mathbb{F}$.

Not all matroids are representable over $\mathbb{F}$. Particularly, in the construction of the normalized matroid (Theorem 3.2) we apply amalgams with (uniform) matroids which need not be $\mathbb{F}$-representable. To achieve their representability, we extend the field $\mathbb{F}$ to an *extension field* $\mathbb{F}' = \mathbb{F}(\alpha)$ with $|\mathbb{F}|^d$ elements in the standard algebraic way with a polynomial root α of degree d.

Lemma 4.1. *The n-element rank-r uniform matroid $U_{r,n}$ is representable over any (finite) field $\mathbb{F}$ such that $|\mathbb{F}| \geq n - 1$.*

Proposition 4.2 (cf. Lemma 2.1). *Let M_1, M_2 be two matroids such that $E(M_1) \cap E(M_2) = T$ and $M_1 \restriction T = M_2 \restriction T$. If both M_1, M_2 are $\mathbb{F}$-represented, and the matroid $M_1 \restriction T$ is free, then there exists an amalgam of M_1 and M_2 which is also $\mathbb{F}$-represented.*

Let k be a fixed integer. We outline a simple new fixed-parameter-tractable algorithm for testing branch-width $\leq k$ on $\mathbb{F}$-represented partitioned matroids:

- First we extend $\mathbb{F}$ to a (nearest) field $\mathbb{F}'$ such that $|\mathbb{F}'| \geq 3k - 6$.
- If, for a given partitioned matroid $(M, \mathcal{P})$, the width of $\mathcal{P}$ is more than k, then the immediate answer is NO.
- Otherwise, we construct the normalized matroid $M^{\#}$ (Theorem 3.2), together with its vector representation over $\mathbb{F}'$ (Lemma 4.1 and Proposition 4.2).
- Finally, we use the algorithm of Hliněný [12] to test the branch-width $\leq k$ of $M^{\#}$.

Theorem 4.3. *Let $k > 1$ be fixed and $\mathbb{F}$ be a finite field. For a partitioned matroid $(M, \mathcal{P})$ represented over $\mathbb{F}$, one can test in time $O(|E(M)|^3)$ (with fixed $k, \mathbb{F}$) whether the branch-width of $(M, \mathcal{P})$ is at most k.*

Now that we are able to test branch-width of partitioned matroids, we show how this result can be extended to finding an appropriate branch-decomposition.

Theorem 4.4. *Let $\mathcal{K}$ be a class of matroids and let k be an integer. If there is an $f(|E(M)|, k)$-time algorithm to decide whether a partitioned matroid $(M, \mathcal{P})$ has branch-width at most k for every pair of a matroid $M \in \mathcal{K}$ and a partition $\mathcal{P}$ of $E(M)$, then a branch-decomposition of the partitioned matroid $(M, \mathcal{P})$ of width at most k, if it exists, can be found in time $O\big(|\mathcal{P}|^3 \cdot f(|E(M)|, k)\big)$.*

The idea of the proof is due to Jim Geelen, published by Oum and Seymour in [20]. We briefly outline the algorithm since it is a base for our improved algorithm in the next section.

- If $|\mathcal{P}| \leq 2$, then it is trivial to output a branch-decomposition.
- We find a pair X, Y of disjoint parts of $\mathcal{P}$ such that a partitioned matroid $(M, (\mathcal{P} \setminus \{X, Y\}) \cup \{X \cup Y\})$ has branch-width at most k. Let $\mathcal{P}' = (\mathcal{P} \setminus \{X, Y\}) \cup \{X \cup Y\}$.
- Let (T', μ') be the branch-decomposition of $(M, \mathcal{P}')$ of width at most k obtained by calling this algorithm recursively.
- Let T be a tree obtained from T' by splitting the leaf $\mu'(X \cup Y)$ into two leaves which we denote by $\mu(X)$ and $\mu(Y)$. Let $\mu(Z) = \mu'(Z)$ for all $Z \in \mathcal{P} \setminus \{X, Y\}$. We output (T, μ) as a branch-decomposition of (M, P) of width at most k.

Corollary 4.5. *For fixed k and finite field $\mathbb{F}$, we can find a branch-decomposition of a given $\mathbb{F}$-represented matroid M of branch-width at most k, if it exists, in time $O(|E(M)|^6)$.*

Remark 4.6. One can actually improve the bound in Theorem 4.4 to $O\big(|\mathcal{P}|^2 \cdot f(|E(M)|, k)\big)$ time. The basic idea is the following: At the first level of recursion we find not only one pair of parts, but a maximal set of disjoint pairs of parts from $\mathcal{P}$ that can be joined (pairwise) while keeping the branch-width at most k. This again requires $O(|\mathcal{P}|^2)$ calls to the decision algorithm. At the deeper levels of recursion we then use the same approach but process only such pairs of parts that contain one joined at the previous level. The details of this approach can be found further in Theorem 5.2.

5 Faster Algorithm for Branch-Decompositions

Even with Remark 4.6 in account, the approach of Section 4 results in an $O(n^5)$ (at best) parametrized algorithm for constructing a branch-decomposition of an n-element matroid represented over a finite field. That is still far from the running time $O(n^3)$ (note fixed k and $\mathbb{F}$) of the decision algorithm in [12]. Although not straightforwardly, we are able to improve the running time of our constructive algorithm to asymptotically match $O(n^3)$ of [12] and [5].

It is the purpose of this section to present a detailed analysis of such a faster implementation of the algorithmic idea of Theorem 4.4 in new Algorithm 5.1. For that we have to dive into fine details of the algorithms in [12], and recall few necessary technical definitions here.

Briefly speaking, a *parse tree* [13] of an $\mathbb{F}$-represented matroid M is a rooted tree $\mathcal{T}$, with at most two children per node, such that: The leaves of $\mathcal{T}$ hold non-loop elements of M represented by points of a projective geometry over $\mathbb{F}$ (or loops of M represented by the empty set). The internal nodes of $\mathcal{T}$, on the other hand, hold *composition operators* over $\mathbb{F}$. A composition operator $\odot$ is a configuration in the projective geometry over $\mathbb{F}$ such that $\odot$ has three subspaces (possibly empty) distinguished as its *boundaries*; two of which are used to "glue" the matroid elements represented in the left and right subtrees, respectively, together. The third one, *upper boundary*, is then used to "glue" this node further up in the parse tree $\mathcal{T}$. (Our "glue" operation, precisely the boundary sum by [13], is analogous to the amalgam of matroids in Proposition 4.2.) The ranks

of adjacent boundaries of two composition operators in $\mathcal{T}$ must be equal for "gluing". A parse tree $\mathcal{T}$ is *$\le t$-boundaried* if all composition operators in $\mathcal{T}$ have boundaries of rank at most t. (Such a parse tree actually gives a branch-decomposition of width at most $t+1$ and vice versa, by [13, Theorem 3.8]).

Algorithm 5.1. Computing a branch-decomposition of a represented partitioned matroid:

Parameters: A finite field $\mathbb{F}$, and a positive integer k.
Input: A rank-r matrix $\boldsymbol{A} \in \mathbb{F}^{r\times n}$ and a partition $\mathcal{P}$ of the columns of $\boldsymbol{A}$. (Assume $n \ge 2$.)
Output: For the vector matroid $M = M(\boldsymbol{A})$ on the columns of $\boldsymbol{A}$, either a branch-decomposition of the partitioned matroid $(M, \mathcal{P})$ of width at most k, or the answer NO if $\mathrm{bw}(M, \mathcal{P}) > k$.

1. Using brute force, we extend the field $\mathbb{F}$ to a (nearest) finite field $\mathbb{F}'$ such that $|\mathbb{F}'| \ge 3k-6$.
2. We check whether $\mathrm{bw}(M, \mathcal{P}) \le k$ (Theorem 4.3, in cubic time). If not, then we answer NO. Otherwise we keep the normalized matroid $M^\#$ and its $\mathbb{F}'$-representation $\boldsymbol{A}^\#$ obtained at this step. We denote by $\mathcal{P}_1$ the (titanic) partition of $E(M^\#)$ corresponding to $\mathcal{P}$, and by $\tau(T) \in \mathcal{P}$ for $T \in \mathcal{P}_1$ the corresponding parts.
3. We compute a $\le 3(k-1)$-boundaried parse tree $\mathcal{T}$ for the matroid $M^\#$ which is $\mathbb{F}'$-represented by $\boldsymbol{A}^\#$ (regardless of $\mathcal{P}_1$). This is done by [12, Algorithm 4.1] in cubic time.
4. We initially set $\mathcal{T}_1 := \mathcal{T}$, $\mathcal{Q}_1 := \emptyset$, $\mathcal{Q}_2 := \big\{\{T_1, T_2\} : T_1 \ne T_2,\ T_1, T_2 \in \mathcal{P}_1\big\}$, and create a new rooted forest D consisting so far of the set of disconnected nodes $\mathcal{P}_1$. Then we repeat the following steps (a),(b), until $\mathcal{P}_1$ contains at most two parts:
   (a) While there is $\{T_1, T_2\} \in \mathcal{Q}_2$ such that $T_1, T_2 \in \mathcal{P}_1$, we do:
      i. Let $\mathcal{Q}_2 := \mathcal{Q}_2 \setminus \big\{\{T_1, T_2\}\big\}$. Calling [12, Algorithm 4.9] in linear time, we compute connectivity $\ell = \lambda_{M_1}(T_1 \cup T_2)$ over the parse tree $\mathcal{T}_1$ which now represents a matroid M_1. If $\ell > k$, then we continue this cycle again from (a).
      ii. We call Algorithm 5.3 on $\mathcal{T}_1$ and $W = T_1 \cup T_2$ to compute a $\le (3k+\ell-2)$-boundaried parse tree $\mathcal{T}_2$. The matroid M_2 of $\mathcal{T}_2$ is actually a represented amalgam (Proposition 4.2) of our *titanic gadget* — a uniform matroid $U_W \simeq U_{\ell-1,3\ell-5}$, with the matroid M_1 (formally replacing W with an $(\ell-1)$-element free sub-matroid of U_W).
      iii. We can immediately check whether branch-width $\mathrm{bw}(M_2) \le k$ by applying [12, Corollary 5.4] on $\mathcal{T}_2$, that is by linear-time testing of the (finitely many by [8]) excluded minors for branch-width at most k. If $\mathrm{bw}(M_2) > k$, then we continue this cycle again from (a).
      iv. So (Lemma 3.1) we have $\mathrm{bw}\big(M_1, \mathcal{P}_1 \cup \{W\} \setminus \{T_1, T_2\}\big) = \mathrm{bw}(M_2) \le k$, and we add a *new node* $E(U_W)$ adjacent to T_1 and T_2 in our constructed decomposition D and make the new node the root for its connected component. We update $\mathcal{P}_1 := \mathcal{P}_2 = \mathcal{P}_1 \cup \{E(U_W)\} \setminus \{T_1, T_2\}$, and $\mathcal{Q}_1 := \mathcal{Q}_1 \cup \{E(U_W)\}$.
      v. Lastly, by calling [12, Algorithm 4.1.3] on $\mathcal{T}_2$, we compute in quadratic time a *new* *$\le 3(k-1)$-boundaried parse tree* $\mathcal{T}_3$ for the matroid M_2, and set $\mathcal{T}_1 := \mathcal{T}_3$.
   (b) When the "while" cycle (4.a) is finished, we set $\mathcal{Q}_2 := \big\{\{T_1, T_2\} : T_1 \ne T_2,\ T_1 \in \mathcal{P}_1,\ T_2 \in \mathcal{Q}_1\big\}$ and $\mathcal{Q}_1 := \emptyset$, and continue from (4.a).

5. Finally, if $|\mathcal{P}_1| = 2$, then we connect by an edge in D the two nodes $T_1, T_2 \in \mathcal{P}_1$. We output (D, τ) as the branch-decomposition of $(M, \mathcal{P})$.

Theorem 5.2. *Let k be a fixed integer and $\mathbb{F}$ be a fixed finite field. We assume that a vector matroid $M = M(\boldsymbol{A})$ is given as an input together with a partition $\mathcal{P}$ of $E(M)$, where $n = |E(M)|$ and $|\mathcal{P}| \geq 2$. Algorithm 5.1 outputs in time $O(n^3)$ (parametrized by k and $\mathbb{F}$), a branch-decomposition of the partitioned matroid $(M, \mathcal{P})$ of width at most k, or confirms that* $\mathrm{bw}(M, \mathcal{P}) > k$.

Note that Algorithm 5.1 implements the general outline of Theorem 4.4. Our proof of this theorem constitutes the following four claims holding true if $\mathrm{bw}(M, \mathcal{P}) \leq k$.

(I) The computation of Algorithm 5.1 maintains invariants, with respect to the actual matroid M_2 of $\mathcal{T}_2$, the decomposition D and current value $\mathcal{P}_2$ of the partition variable $\mathcal{P}_1$ after each call to step (4.a.iv), that
- $\mathcal{P}_2$ is the set of roots of D, and a titanic partition of M_2 such that $\mathrm{bw}(M_2, \mathcal{P}_2) = \mathrm{bw}(M_2) \leq k$,
- $\lambda_M\big(\tau^D(\mathcal{S})\big) = \lambda^{\mathcal{P}_2}_{M_2}(\mathcal{S})$ for each $\mathcal{S} \subseteq \mathcal{P}_2$, where $\tau^D(\mathcal{S})$ is a shortcut for the union of $\tau(T)$ with T running over all leaves of the connected components of D whose root is in $\mathcal{S}$ (see Algorithm 5.1 step 2. for τ).

(II) Each iteration of the main cycle in Algorithm 5.1(4.) succeeds to step (4.a.iv) at least once.

(III) The main cycle in Algorithm 5.1(4.) is repeated $O(n)$ times. Moreover, the total number of calls to the steps in (4.a) is: $O(n^2)$ for steps i,ii,iii, and $O(n)$ for steps iv,v.

(IV) Further defined Algorithm 5.3 (cf. step (4.a.ii)) computes correctly in time $O(n)$.

Having all these facts at hand, it is now easy to finish the proof. It is immediate from (I) that resulting (D, τ) is a branch-decomposition of width at most k of $(M, \mathcal{P})$. Note that all parse trees involved in the algorithm have constant width less than $4k$ (see in steps (4.a.ii,v)). The starting steps (1.),(2.),(3.) of the algorithm are already known to run in time $O(n^3)$ (Hliněný [12] and Theorem 4.3), and the particular steps in (4.a) need time $O(n^2) \cdot O(n) + O(n) \cdot O(n^2) = O(n^3)$ by (III) and (IV).

We are left with (IV), an immediate extension of [12, Algorithm 4.9] computing $\lambda_{M_1}(W)$.

Algorithm 5.3. Computing an amalgam with a uniform matroid on the parse tree.

Input: A $\leq(3k-1)$-boundaried parse tree $\mathcal{T}_1$ of a matroid M_1, and a set $W \subseteq E(M_1)$ such that $\lambda_{M_1}(W) = \ell \leq k$.

Output: A $\leq(3k+\ell-2)$-boundaried parse tree $\mathcal{T}_2$ representing a matroid M_2 on the ground set $E(M_2) = E_1 \cup E_2$ where $E_1 = E(M_1) \setminus W$ and $E_2 \cap E(M_1) = \emptyset$; such that $M_2 \restriction E_1 = M_1 \restriction E_1$, $M_2 \restriction E_2 = U_W \simeq U_{\ell-1,3\ell-5}$, and the set $E_2 = E(U_W)$ is spanned both by E_1 and by W in the (combined) point configuration $M_1 \cup M_2$.

6 Finding a Rank-Decomposition of a Graph

In this last section, we present a fixed-parameter-tractable algorithm to find a rank-decomposition of width at most k or confirm that the input graph has rank-width larger than k. It is a direct translation of the algorithm of Theorem 5.2. Let us first review necessary definitions. We assume that all graphs in this section have no loops and no parallel edges.

We have seen in Section 2 that every symmetric submodular function can be used to define branch-width. We define a symmetric submodular function on a graph, called the *cut-rank* function of a graph. For an $X \times Y$ matrix $\boldsymbol{R}$ and $A \subseteq X$, $B \subseteq Y$, let $\boldsymbol{R}[A, B]$ be the $A \times B$ submatrix of $\boldsymbol{R}$. For a graph G, let $\boldsymbol{A}(G)$ be the adjacency matrix of G, that is a $V \times V$ matrix over the binary field GF(2) such that an entry is 1 if and only if vertices corresponding to the column and the row are adjacent in G. The cut-rank function $\rho_G(X)$ of a graph $G = (V, E)$ is defined as the rank of the matrix $\boldsymbol{A}(G)[X, V \setminus X]$ for each subset X of V. Then ρ_G is symmetric and submodular, see [19]. *Rank-decomposition* and *rank-width* and of a graph G is branch-decomposition and branch-width of the cut-rank function ρ_G of the graph G, respectively. So if the graph has at least two vertices, then the rank-width is at most k if and only if there is a rank-decomposition of width at most k.

Now let us recall why bipartite graphs are essentially binary matroids. Oum [17] showed that the connectivity function of a binary matroid is exactly one more than the cut-rank function of its *fundamental graph*. The fundamental graph of a binary matroid M on $E = E(M)$ with respect to a basis B is a bipartite graph on E such that two vertices in E are adjacent if and only if one vertex v is in B, another vertex w is not in B, and $(B \setminus \{v\}) \cup \{w\}$ is independent in M. Given a bipartite graph G, we can easily construct a binary matroid having G as a fundamental graph; if (C, D) is a bipartition of $V(G)$, then take the matrix

$$\left(\begin{array}{ccc|c} 1 & & 0 & \boldsymbol{A}(G)[C,D] \\ & \ddots & & C \times D \text{ submatrix of} \\ 0 & & 1 & \text{the adjacency matrix} \end{array}\right)$$

as the representation of a binary matroid. (Thus the column indices are elements of the binary matroid and a set of columns is independent in the matroid if and only if its vectors are linearly independent.) After all, finding the rank-decomposition of a bipartite graph is equivalent to finding the branch-decomposition of the associated binary matroid, that is essentially Theorem 5.2.

To find a rank-decomposition of non-bipartite graphs, we transform the graph into a canonical bipartite graph. For a finite set V, let V^* be a disjoint copy of V, that is, formally speaking, $V^* = \{v^* : v \in V\}$ such that $v^* \neq w$ for all $w \in V$ and $v^* \neq w^*$ for all $w \in V \setminus \{v\}$. For a subset X of V, let $X^* = \{v^* : v \in X\}$. For a graph $G = (V, E)$, let $\mathrm{bip}(G)$ be the bipartite graph on $V \cup V^*$ such that vw^* are adjacent in $\mathrm{bip}(G)$ if and only if v and w are adjacent in G. Let $P_v = \{v, v^*\}$ for each $v \in V$. Then $\Pi(G) = \{P_v : v \in V\}$ is a canonical partition of $V(\mathrm{bip}(G))$.

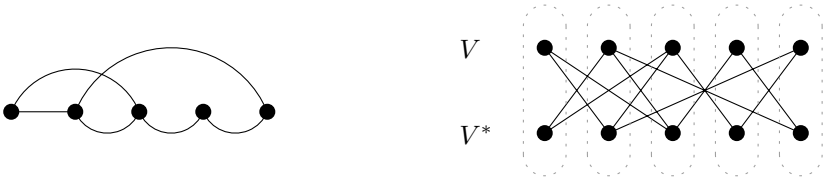


Fig. 2. Graph G and the associated bipartite graph $\mathrm{bip}(G)$ with its canonical partition

Lemma 6.1. *For every subset X of $V(G)$, $2\rho_G(X) = \rho_{\mathrm{bip}(G)}(X \cup X^*)$.*

Corollary 6.2. *Let $p : V(G) \to \Pi(G)$ be the bijective function such that $p(x) = P_x$. If (T, μ) is a branch-decomposition of $\rho^{\Pi(G)}_{\mathrm{bip}(G)}$ of width k, then $(T, \mu \circ p)$ is a branch-decomposition of ρ_G of width $k/2$. Conversely, if (T, μ') is a branch-decomposition of ρ_G of width k, then $(T, \mu' \circ p^{-1})$ is a branch-decomposition of $\rho^{\Pi(G)}_{\mathrm{bip}(G)}$ of width $2k$. Therefore the branch-width of ρ_G is equal to the half of the branch-width of $\rho^{\Pi(G)}_{\mathrm{bip}(G)}$.*

Let $M = \mathrm{mat}(G)$ be the binary matroid on $V \cup V^*$ represented by the matrix:

$$\begin{array}{c} \\ V \end{array} \begin{array}{c} \begin{array}{cc} V & V^* \end{array} \\ \left(\begin{array}{c|c} \text{Identity matrix} & \boldsymbol{A}(G) \end{array}\right). \end{array}$$

Since the bipartite graph $\mathrm{bip}(G)$ is a fundamental graph of M, we have $\lambda_M(X) = \rho_{\mathrm{bip}(G)}(X)+1$ for all $X \subseteq V \cup V^*$ (see Oum [17]) and therefore (T, μ) is a branch-decomposition of a partitioned matroid $(M, \Pi(G))$ of width $k+1$ if and only if it is a branch-decomposition of $\rho^{\Pi(G)}_{\mathrm{bip}(G)}$ of width k. Corollary 6.2 implies that a branch-decomposition of $\rho^{\Pi(G)}_{\mathrm{bip}(G)}$ of width k is equivalent to that of ρ_G of width $k/2$. So we can deduce the following theorem from Theorem 5.2.

Theorem 6.3. *Let k be a constant. Let $n \geq 2$. For an n-vertex graph G, we can output the rank-decomposition of width at most k or confirm that the rank-width of G is larger than k in time $O(n^3)$.*

Acknowledgments. The authors are very grateful to Jim Geelen for his comments on the possible approach to the problem. We would also like to thank the anonymous referees for helpful comments.

References

1. Corneil, D.G., Habib, M., Lanlignel, J.M., Reed, B., Rotics, U.: Polynomial time recognition of clique-width ≤ 3 graphs (extended abstract). In: Gonnet, G.H., et al. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 126–134. Springer, Heidelberg (2000)

2. Corneil, D.G., Perl, Y., Stewart, L.K: A linear recognition algorithm for cographs. SIAM J. Comput. 14(4), 926–934 (1985)
3. Courcelle, B., Makowsky, J.A., Rotics, U.: Linear time solvable optimization problems on graphs of bounded clique-width. Theory Comput. Syst. 33(2), 125–150 (2000)
4. Courcelle, B., Olariu, S.: Upper bounds to the clique width of graphs. Discrete Appl. Math. 101(1-3), 77–114 (2000)
5. Courcelle, B., Oum, S.: Vertex-minors, monadic second-order logic, and a conjecture by Seese. J. Combin. Theory Ser. B 97(1), 91–126 (2007)
6. Espelage, W., Gurski, F., Wanke, E.: How to solve NP-hard graph problems on clique-width bounded graphs in polynomial time. In: Brandstädt, A., Le, V.B. (eds.) WG 2001. LNCS, vol. 2204, Springer, Heidelberg (2001)
7. Fellows, M.R., Rosamond, F.A., Rotics, U., Szeider, S.: Clique-width minimization is NP-hard. In: Proceedings of the 38th annual ACM Symposium on Theory of Computing, pp. 354–362. ACM Press, New York, USA (2006)
8. Geelen, J.F., Gerards, A.M.H., Robertson, N., Whittle, G.: On the excluded minors for the matroids of branch-width k. J. Combin. Theory Ser. B 88(2), 261–265 (2003)
9. Geelen, J.F., Gerards, A.M.H., Whittle, G.: Tangles, tree-decompositions, and grids in matroids. Research Report 04-5, School of Mathematical and Computing Sciences, Victoria University of Wellington (2004)
10. Gerber, M.U., Kobler, D.: Algorithms for vertex-partitioning problems on graphs with fixed clique-width. Theoret. Comput. Sci. 299(1-3), 719–734 (2003)
11. Hicks, I.V., McMurray, Jr., N.B.: The branchwidth of graphs and their cycle matroids. J. Combin. Theory Ser. B, 97(5), 681–692, (2007)
12. Hliněný, P.: A parametrized algorithm for matroid branch-width (loose erratum (electronic)). SIAM J. Comput. 35(2), 259–277 (2005)
13. Hliněný, P.: Branch-width, parse trees, and monadic second-order logic for matroids. J. Combin. Theory Ser. B 96(3), 325–351 (2006)
14. Kobler, D., Rotics, U.: Edge dominating set and colorings on graphs with fixed clique-width. Discrete Appl. Math. 126(2-3), 197–221 (2003)
15. Mazoit, F., Thomassé, S.: Branchwidth of graphic matroids. Manuscript (2005)
16. Oum, S.: Approximating rank-width and clique-width quickly. In: Kratsch, D. (ed.) WG 2005. LNCS, vol. 3787, pp. 49–58. Springer, Heidelberg (2005)
17. Oum, S.: Rank-width and vertex-minors. J. Combin. Theory Ser. B 95(1), 79–100 (2005)
18. Oum, S.: Approximating rank-width and clique-width quickly. Submitted, an extended abstract appeared in [16] (2006)
19. Oum, S., Seymour, P.: Approximating clique-width and branch-width. J. Combin. Theory Ser. B 96(4), 514–528 (2006)
20. Oum, S., Seymour, P.: Testing branch-width. J. Combin. Theory Ser. B 97(3), 385–393 (2007)
21. Oxley, J.G.: Matroid theory. Oxford University Press, New York (1992)
22. Robertson, N., Seymour, P.: Graph minors. X. Obstructions to tree-decomposition. J. Combin. Theory Ser. B 52(2), 153–190 (1991)
23. Seymour, P., Thomas, R.: Call routing and the ratcatcher. Combinatorica 14(2), 217–241 (1994)
24. Wanke, E.: k-NLC graphs and polynomial algorithms. Discrete Appl. Math. 54(2-3), 251–266 (1994)

Fast Algorithms for Maximum Subset Matching and All-Pairs Shortest Paths in Graphs with a (Not So) Small Vertex Cover

Noga Alon[1] and Raphael Yuster[2]

[1] School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
nogaa@post.tau.ac.il
[2] Department of Mathematics, University of Haifa, Haifa, Israel
raphy@math.haifa.ac.il

Abstract. In the *Maximum Subset Matching* problem, which generalizes the maximum matching problem, we are given a graph $G = (V, E)$ and $S \subset V$. The goal is to determine the maximum number of vertices of S that can be matched in a matching of G. Our first result is a new randomized algorithm for the Maximum Subset Matching problem that improves upon the fastest known algorithms for this problem. Our algorithm runs in $\tilde{O}(ms^{(\omega-1)/2})$ time if $m \geq s^{(\omega+1)/2}$ and in $\tilde{O}(s^{\omega})$ time if $m \leq s^{(\omega+1)/2}$, where $\omega < 2.376$ is the matrix multiplication exponent, m is the number of edges from S to $V \setminus S$, and $s = |S|$. The algorithm is based, in part, on a method for computing the rank of sparse rectangular integer matrices.

Our second result is a new algorithm for the *All-Pairs Shortest Paths* (APSP) problem. Given an undirected graph with n vertices, and with integer weights from $\{1, \ldots, W\}$ assigned to its edges, we present an algorithm that solves the APSP problem in $\tilde{O}(Wn^{\omega(1,1,\mu)})$ time where $n^{\mu} = vc(G)$ is the *vertex cover number* of G and $\omega(1, 1, \mu)$ is the time needed to compute the Boolean product of an $n \times n$ matrix with an $n \times n^{\mu}$ matrix. Already for the unweighted case this improves upon the previous $O(n^{2+\mu})$ and $\tilde{O}(n^{\omega})$ time algorithms for this problem. In particular, if a graph has a vertex cover of size $O(n^{0.29})$ then APSP in unweighted graphs can be solved in asymptotically optimal $\tilde{O}(n^2)$ time, and otherwise it can be solved in $O(n^{1.844} vc(G)^{0.533})$ time.

The common feature of both results is their use of algorithms developed in recent years for fast (sparse) rectangular matrix multiplication.

1 Introduction

A *matching* in a graph is a set of pairwise disjoint edges. A *maximum matching* is a matching of largest possible size. The problem of finding a maximum matching is fundamental in both practical and theoretical computer science.

The first polynomial time algorithm for finding a maximum matching in a general graph was obtained by Edmonds [6]. The currently fastest deterministic algorithms for this problem run in $O(mn^{1/2})$ time (see [13,2,19,7]), where m and

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 175–186, 2007.

n are the number of edges and vertices, respectively, in the input graph. For dense graphs, better *randomized* algorithms are known. Lovász [12] showed that the cardinality of a maximum matching can be determined, with high probability, by computing the rank of a matrix. In particular, checking whether a graph has a perfect matching amounts to checking whether a determinant of a certain matrix, whose construction involves randomization, is nonzero. This randomized algorithm can be implemented to run in $O(n^\omega)$ time, where ω is the exponent of fast matrix multiplication. Coppersmith and Winograd [5] showed that $\omega < 2.376$. Recently, Mucha and Sankowski [14] solved a long standing open problem and showed that a maximum matching can be *found*, with high probability, in $O(n^\omega)$ time.

Our first main result in this paper concerns a natural generalization of the maximum matching problem. For a graph $G = (V, E)$ and a subset of vertices $S \subset V$, let $f(S)$ denote the maximum possible number of vertices in S that can be matched in a matching of G. Notice that if $S = V$ then $f(S)$ is simply the size of a maximum matching of G. In general, however, not every maximum matching of G saturates the maximum possible number of vertices of S (e.g. consider a triangle where S consists of two vertices) and, conversely, not every matching that saturates $f(S)$ vertices of S can be extended to a maximum matching (e.g. consider a path of length 3 where S consists of the two internal vertices). Thus, the *MAXIMUM SUBSET MATCHING* problem is a true generalization of the maximum matching problem.

Maximum Subset Matching also has natural applications. Consider a graph modeling a social network, the vertices being the members of the network and the edges being the symmetric social relations. There are two types of members: privileged members (e.g. registered customers) and non-privileged members. Our goal is to socially match the maximum number of privileged members.

We present an efficient randomized algorithm for computing $f(S)$. The running time of our algorithm is expressed in terms of $s = |S|$ and the number of edges connecting S to $V \setminus S$, denoted by m. There could be as many as $\Theta(s^2)$ edges inside S, and hence the graph may have as many as $\Theta(m + s^2)$ edges (clearly we may assume that $V \setminus S$ is an independent set). As usual in matching algorithms, we assume that the graph has no isolated vertices.

Theorem 1. *Let $G = (V, E)$ be a graph and let $S \subset V$, where $|S| = s$ and there are m edges from S to $V \setminus S$. Then, there is a randomized algorithm that computes $f(S)$ w.h.p. in time*

$$\tilde{O}\left(\begin{cases} ms^{\frac{\omega-1}{2}} & \text{if } m \geq s^{\frac{\omega+1}{2}}, \\ s^\omega & \text{if } m \leq s^{\frac{\omega+1}{2}} \end{cases}\right).$$

To evaluate the running time obtained in Theorem 1 we compare our algorithm to known existing matching algorithms. First notice that for the current best upper bound of ω, which is less than 2.376, we have that if $m \geq s^{1.688}$ the algorithm runs in $O(ms^{0.688})$ time, and if $m \leq s^{1.688}$ the algorithm runs in $O(s^{2.376})$ time. Notice that the running time is *not* expressed as a function of

$|V| = n$ and that n may be as large as $s+m$. Maximum Subset Matching can be solved via maximum *weighted* matching algorithms as follows. Assign to every vertex with both endpoints in S weight 2, and edges from S to $V \setminus S$ weight 1. Then, clearly, a maximum weighted matching has weight $f(S)$. The algorithm of Gabow and Tarjan [7] is currently the fastest algorithm for maximum weighted matching in general graphs. In our setting, it runs in $\tilde{O}(\sqrt{n}(m+s^2))$ time (here $m+s^2$ is our upper bound on the total number of edges), which is worse than the running time of the algorithm of Theorem 1 for a wide range of parameters (in fact, if $\omega = 2 + o(1)$ as conjectured by many researchers, then it is never better, for any valid combination of the parameters s, m, n). It is not known how to apply Lovás'z randomized maximum cardinality matching algorithm to the weighted case. Even if it were possible, the obtained running time would only be $\tilde{O}(n^\omega)$.

We now turn to our second main result. In the *All-Pairs Shortest Paths* (APSP) problem, which is one of the most fundamental algorithmic graph problems, we are given a graph $G = (V, E)$ with $V = \{1, \ldots, n\}$, and our goal is to compute the *distance matrix* of G. This is an $n \times n$ matrix D where $D(i, j)$ is the length of a shortest path from i to j, for all $i, j = 1, \ldots, n$. We also require a concise $n \times n$ data structure so that given a pair i, j, a shortest path from i to j can be constructed in time which is proportional to the number of edges it contains. In the most fundamental case, the graph is undirected and unweighted; this is the main case we address in this paper (more generally, we allow the weights to be positive integers). The currently fastest algorithms for the APSP problem are either the simple obvious $O(nm)$ algorithm (here $m = |E|$) consisting of breadth first search from each vertex, or, as m gets larger, a randomized algorithm of Seidel [16] that solves the problem in $\tilde{O}(n^\omega)$ time. This algorithm also has a deterministic version [8,1]. Shoshan and Zwick [17] proved that if the graph has positive weights from $\{1, \ldots, W\}$ then APSP can be solved, deterministically, in $\tilde{O}(Wn^\omega)$ time.

Is there a natural graph parameter for which there exists an algorithm whose running time is expressed in terms of it and which always performs at least as well as the $\tilde{O}(n^\omega)$ algorithm, and is generally faster? Our main result shows that there is one.

Let $vc(G)$ be the *vertex cover number* of G, that is the smallest possible cardinality of a subset $S \subset V$ so that $V \setminus S$ is an independent set. The fact that $vc(G)$ is NP-Hard to compute is well known and its decision version is one of the canonical examples for NP-Completeness. It is also well known, and trivial, that $vc(G)$ can be approximated to within a factor of two, in linear time (simply take all the vertices in a maximal matching with respect to containment). Our main result shows that if G is an undirected graph with n vertices and $vc(G) = n^\mu$ then the APSP problem can be solved in time $\tilde{O}(Wn^{\omega(1,1,\mu)})$, where each weight is taken from $\{1, \ldots, W\}$. Here $\omega(1, 1, \mu)$ is the rectangular matrix multiplication exponent. Namely, it is the number of algebraic operations needed to multiply an $n \times n$ matrix with an $n \times n^\mu$ matrix over an arbitrary ring (hence, $\omega = \omega(1, 1, 1)$).

Theorem 2. *There is an APSP algorithm that, given an undirected n-vertex graph G with weights from $\{1,\ldots,W\}$ and $vc(G) = n^{\mu}$, runs in $\tilde{O}(Wn^{\omega(1,1,\mu)})$ time. In particular, if W is constant and $vc(G) \leq n^{0.294}$ then the algorithm runs in asymptotically optimal $\tilde{O}(n^2)$ time and, if $vc(G) > n^{0.294}$ then the algorithm runs in $O(n^{1.844}vc(G)^{0.533})$ time.*

In our proof of Theorem 2 we construct the distance matrix in the guaranteed running time. It is also possible to obtain a concise data structure representing the paths, by using witnesses for (rectangular) matrix multiplication, as shown in [1]; the details of this latter construction, which is quite standard, will appear in the full version of this paper. In Section 2 we list the known results for $\omega(1,1,\mu)$. In particular, Coppersmith [4] proved that $\omega(1,1,\mu) = 2 + o(1)$ for $\mu < 0.294$, and that, assuming $\omega = \omega(1,1,1) > 2$, then $\omega(1,1,\mu) < \omega(1,1,1)$ for all $\mu < 1$ (see Lemma 2 in the next section). Thus, at present, our algorithm is faster than the $O(n^{\omega})$ algorithm for all unweighted graphs having a vertex cover of size $n^{1-\epsilon}$. We also note that the proof of Theorem 2 does not assume that we can compute $vc(G)$ precisely. As mentioned earlier, there is a $\Theta(m+n)$ 2-approximation algorithm for $vc(G)$, which suffices for our purposes.

The rest of this paper is organized as follows. As both of our main results, although having completely different proofs, rely on fast rectangular matrix multiplication, we present in Section 2 the facts we need about fast rectangular matrix multiplication algorithms. In Section 3 we address our first main result on the maximum subset matching problem and prove Theorem 1. The APSP problem in undirected graphs is addressed in Section 4 where Theorem 2 is proved. The final section contains some concluding remarks and open problems.

2 Fast (Sparse) Rectangular Matrix Multiplication

We start this section by presenting some parameters related to fast rectangular matrix multiplication. Our assumption is that the input matrices are given to us in sparse representation, that is as a collection of triplets (row, column, value) listing all positions where the matrix is non-zero. Thus, the size of the representation of an input matrix is linear in the number of non-zero entries of the matrix.

Let $M(a,b,c)$ be the minimal number of algebraic operations needed to multiply an $a \times b$ matrix by a $b \times c$ matrix over an arbitrary ring R. Let $\omega(r,s,t)$ be the minimal exponent for which $M(n^r,n^s,n^t) = O(n^{\omega(r,s,t)})$. Recall that $\omega = \omega(1,1,1) < 2.376$ [5]. The best bounds available on $\omega(1,\mu,1)$, for $0 \leq \mu \leq 1$ are summarized in the following results. Before stating them we need to define two more constants, α and β, related to rectangular matrix multiplication.

Definition 1. $\alpha = \max\{0 \leq \mu \leq 1 \mid \omega(1,\mu,1) = 2 + o(1)\}$, $\beta = \dfrac{\omega - 2}{1 - \alpha}$.

Lemma 1 (Coppersmith [4]). $\alpha > 0.294$.

It is not difficult to see that Lemma 1 implies the following theorem. A proof can be found, for example, in Huang and Pan [10].

Lemma 2

$$\omega(1,\mu,1)=\omega(1,1,\mu)=\omega(\mu,1,1) \leq \begin{cases} 2+o(1) & \text{if } 0\leq\mu\leq\alpha, \\ 2+\beta(\mu-\alpha)+o(1) & \text{otherwise.} \end{cases}$$

Notice that Coppersmith's result implies that if A is an $n\times n^{0.294}$ matrix and B is an $n^{0.294}\times n$ matrix then AB can be computed in $O(n^{2+o(1)})$ time (assuming ring operations take constant time), which is essentially optimal since the product is an $n\times n$ matrix that may contain no zero elements at all. Note that with $\omega=2.376$ and $\alpha=0.294$ we get $\beta\simeq 0.533$. If $\omega=2+o(1)$, as conjectured by many, then $\alpha=1$. (In this case β is not defined, but also not needed).

We now present results for fast sparse matrix multiplication. The running times of these algorithms are expressed in terms of α, β, and ω.

Lemma 3 (Yuster and Zwick [21]). *The product of two $n\times n$ matrices over a ring R, each with at most m non-zero elements, can be computed in time*

$$O(\min\{\ m^{\frac{2\beta}{\beta+1}}n^{\frac{2-\alpha\beta}{\beta+1}}+n^{2+o(1)}\ ,\ n^{\omega}\ \}).$$

Note that for $\omega=2.376$, $\alpha=0.294$, and $\beta=0.533$, the algorithm runs in $O(\min\{m^{0.7}n^{1.2}+n^{2+o(1)}\ ,\ n^{2.376}\})$ time (in fact, by "time" we assume that each algebraic operation in the ring takes constant time; if not, then the running time should be multiplied by the time needed for an algebraic operation).

It is possible to extend the method of [21] to rectangular matrices. This was explicitly done in [11]. To simplify the runtime expressions, we only state the case where A and B^T (where A and B are the two matrices being multiplied) have the same dimensions, as this case suffices for our purposes.

Lemma 4 (Kaplan, Sharir, and Verbin [11]). *Let A be an $n\times r$ matrix and let B be an $r\times n$ matrix, over a ring R, each with at most m non-zero elements. Then AB can be computed in time*

$$O\left(\begin{cases} mn^{\frac{\omega-1}{2}} & \text{if } m\geq n^{\frac{\omega+1}{2}}, \\ m^{\frac{2\beta}{\beta+1}}n^{\frac{2-\alpha\beta}{\beta+1}} & \text{if } n^{1+\frac{\alpha}{2}}\leq m\leq n^{\frac{\omega+1}{2}}, \\ n^{2+o(1)} & \text{if } m\leq n^{1+\frac{\alpha}{2}} \end{cases}\right).$$

Notice that the value of r in the statement of Theorem 4 is irrelevant since our sparse representation (and the fact that a common all-zero column of A and B^T can be discarded without affecting the product) implies that $r=O(m)$.

Finally, a word about Boolean matrix multiplication. Although not properly a ring, all of the results in this section also apply to Boolean matrix multiplication. This is easy to see; simply perform the operations over Z_{n+1}. A non-zero value is interpreted as 1. As each algebraic operation in Z_n costs only $\Theta(\log n)$ time it is not surprising that some researchers actually define ω as the exponent of Boolean matrix multiplication.

3 Maximum Subset Matching

Let $G = (V, E)$ be an undirected graph with $V = \{1, \ldots, n\}$. With each edge $e \in E$ we associate a variable x_e. Define the *skew adjacency matrix* (also known as the Tutte matrix) $A_t(G) = (a_{ij})$ by

$$a_{ij} = \begin{cases} +x_e, & \text{if } e = ij,\ i < j \text{ and } (i,j) \in E; \\ -x_e, & \text{if } e = ij,\ i > j \text{ and } (i,j) \in E; \\ 0, & \text{otherwise.} \end{cases}$$

Tutte [18] showed that $A_t(G)$ is non-singular if and only if G has a perfect matching. This was generalized later by Lovász [12] who proved that:

Lemma 5. *The size of a maximum matching in G is $\frac{1}{2}rank(A_t(G))$.*

This fact, together with some additional non-trivial ideas, leads to an $O(n^\omega)$ time randomized algorithm for deciding whether a graph has a perfect matching, and, much later, to $O(n^\omega)$ time randomized algorithms that actually *find* a maximum matching [14,9].

Our first lemma is a generalization of Lemma 5. For a graph $G = (V, E)$ and a subset $S \subset V$, let $A_t(S, G)$ denote the sub-matrix of $A_t(G)$ obtained by taking only the rows corresponding to vertices of S. Thus, $A_t(S, G)$ is an $s \times n$ matrix where $s = |S|$ and $n = |V|$. Recalling the definition of $f(S)$ from the introduction we prove:

Lemma 6

$$f(S) = rank(A_t(S, G)).$$

Proof. We first prove that $f(S) \leq rank(A_t(S, G))$. Consider the case where $s - f(S)$ is even (the odd case is proved analogously). We may assume that $n - s$ is even since otherwise, we can alway add an additional isolated vertex to $V \setminus S$ without affecting the rank of $A_t(S, G)$ nor the value $f(S)$. We add edges to G so that $V \setminus S$ induces a complete graph, and denote the new graph by G'. Clearly, the cardinality of the maximum matching of G' is $(n - s + f(S))/2$. Thus, by Lemma 5, $rank(A_t(G')) = n - s + f(S)$. In particular, we must have that

$$rank(A_t(S, G')) \geq rank(A_t(G')) - (n - s) = f(S).$$

But $A_t(S, G')$ and $A_t(S, G)$ are the same matrix, hence $rank(A_t(S, G)) \geq f(S)$.

We next prove that $f(S) \geq rank(A_t(S, G))$. Suppose that $rank(A_t(S, G)) = r$. We have to show that at least r vertices of S can be saturated by a matching. As the rank is r there is an $r \times r$ sub-matrix of $A_t(S, G))$, call it B, which is nonsingular. Let the rows of B correspond to S' (notice that $S' \subset S$), and the columns to U, which is some set of vertices, possibly intersecting S'. It suffices to show that there is a matching covering the vertices in S'. In the expansion of the determinant of B we get $r!$ products (with signs). Each product corresponds to an oriented subgraph of G obtained by orienting, for each x_{ij} in the product, the edge from i (the row) to j (the column). This gives a subgraph in which

the out-degree of every vertex of S' is 1 and the indegree of every vertex of U is 1. Thus any connected component is either a directed path, all of whose vertices are in S' besides the last one which is in $U \setminus S'$, or a cycle, all of whose vertices are in S'. The crucial point now is that if there is an odd cycle in this subgraph, then the contribution of this term to the determinant is zero, as we can orient the cycle backwards and get the same term with an opposite sign (we do it only for the lexicographically first such cycle in the subgraph, to make sure this is well defined; this will pair the terms that cancel). As the determinant is nonzero, there is at least one such subgraph in which all components are either paths or even cycles, and hence there is a matching saturating all vertices in S', as needed. □

Our algorithm computes $f(S)$ by computing $rank(A_t(S,G))$ with high probability which, by Lemma 6, amounts to the same thing. Computing the rank of a symbolic matrix (such as $A_t(S,G)$) directly is costly. Each algebraic operation is performed in a ring of multivariate polynomials of high degree, and cannot, therefore, be performed in constant (or, close to constant) time. By using a result of Zippel [23] and Schwartz [15], Lovász [12] proved the following.

Lemma 7. *If G is a graph with n vertices and we replace each variable of $A_t(G)$ with a random integer from $\{1,\ldots,R\}$ then the rank of the resulting matrix equals $rank(A_t(G))$ with probability at least $1-n/R$. Similarly, if B is any given sub-matrix of $A_t(G)$ then the rank of the resulting sub-matrix equals the rank of B, with probability at least $1-n/R$.*

For a complex matrix A, let A^* denote, as usual, the Hermitian transpose of A (some researchers also denote it by A^H). If A is a real matrix then $A^* = A^T$. We need to recall the following simple fact from linear algebra.

Fact 8. *Let A be a complex matrix, then A^*A and A have the same kernel.*

Indeed, suppose $A^*Ax = 0$, then, using the Hermitian product, $< Ax, Ax >= < A^*Ax, x >= 0$, whence $Ax = 0$. Notice that the assertion may fail for general fields, as can be seen, for instance, by the $p \times p$ matrix over F_p, all of whose entries are equal to 1.

We need the following result of Hopcroft and Bunch [3] which asserts that Gaussian Elimination of a matrix requires asymptotically the same number of algebraic operations needed for matrix multiplication. Notice that a by-product of Gaussian elimination is the matrix rank.

Lemma 9. *Let A be an $n \times n$ matrix over an arbitrary field. Then $rank(A)$ can be computed using $O(n^\omega)$ algebraic operations. In particular, if each field operation requires $\Theta(K)$ time then $rank(A)$ can be computed in $O(Kn^\omega)$ time.*

An important proposition which is obtained by combining Lemma 9, Lemma 4, Fact 8, and one additional idea, is the following:

Theorem 3. *Let A be an $s \times n$ matrix having at most m non-zero integer entries located in an $s \times (n-s)$ sub-matrix (the other s^2 entries may be all non-zero),*

and suppose that the largest absolute value of an entry is R. Then, $rank(A)$ can be computed, w.h.p., in time

$$\tilde{O}\left(\left\{\begin{array}{ll}(\log R)ms^{\frac{\omega-1}{2}} & if\ m \geq s^{\frac{\omega+1}{2}},\\ (\log R)s^{\omega} & if\ m \leq s^{\frac{\omega+1}{2}}\end{array}\right.\right).$$

Proof: Let A_2 be the $s \times (n-s)$ sub-matrix containing at most m non-zero entries, and let A_1 be the remaining $s \times s$ sub-matrix. Clearly,

$$B = AA^T = A_1A_1^T + A_2A_2^T.$$

We first compute $A_1A_1^T$ using $O(s^{\omega})$ algebraic operations and in $O((\log R)s^{\omega})$ time, as each algebraic operation requires $O(\log R)$ time. We compute $A_2A_2^T$ using Lemma 4. This can be done in the time stated in Lemma 4, multiplied by $\log R$. We have therefore computed B. Notice that B is an $s \times s$ matrix, and each entry in B has absolute value at most nR^2. Furthermore, by Fact 8, $rank(B) = rank(A)$. Now, suppose $rank(B) = t$. Thus, B has a $t \times t$ sub-matrix B' whose rank is t, and hence $\det(B') \neq 0$. On the other hand, by Hadamard Inequality $|det(B')| \leq (tn^2R^4)^{t/2} < (nR)^{2n}$. Since an integer x has $O(\log x)$ prime divisors, choosing a random prime $p = O((nR)^2)$ guarantees that, w.h.p., $det(B') \neq 0$ also in F_p, and, in particular, $rank(B) = t$ also in F_p. We compute $rank(B)$ using Lemma 9, using $O(s^{\omega})$ algebraic operations, where each algebraic operation in F_p requires $O(\log n + \log r)$ time. Thus, in time $\tilde{O}((\log R)s^{\omega})$. The overall running time of the algorithm is

$$\begin{aligned}&\tilde{O}\left(\left\{\begin{array}{ll}(\log R)(s^{\omega} + ms^{\frac{\omega-1}{2}}) & \text{if } m \geq s^{\frac{\omega+1}{2}},\\ (\log R)(s^{\omega} + m^{\frac{2\beta}{\beta+1}}s^{\frac{2-\alpha\beta}{\beta+1}}) & \text{if } s^{1+\frac{\alpha}{2}} \leq m \leq s^{\frac{\omega+1}{2}},\\ (\log R)(s^{\omega} + s^{2+o(1)}) & \text{if } m \leq s^{1+\frac{\alpha}{2}}\end{array}\right.\right)\\ &= \tilde{O}\left(\left\{\begin{array}{ll}(\log R)ms^{\frac{\omega-1}{2}} & \text{if } m \geq s^{\frac{\omega+1}{2}},\\ (\log R)s^{\omega} & \text{if } m \leq s^{\frac{\omega+1}{2}}\end{array}\right.\right).\end{aligned}$$

□

Completing the Proof of Theorem 1: We are given a graph $G = (V, E)$ and a subset $S \subset V$, where $|S| = s$ and there are m edges between S and $V \setminus S$. Our goal is to compute $f(S)$.

We construct the Tutte sub-matrix $A_t(S, G)$ and replace each variable with an integer from $\{1, \ldots, n^2\}$, uniformly at random. Denote the obtained integer matrix by A. By Lemma 7, with probability at least $1 - 1/n$, $rank(A) = rank(A_t(S, G))$. Thus, we need to compute $rank(A)$. Notice, however, that A is an $s \times n$ integer matrix with at most m non-zero entries in an $s \times (n-s)$ sub-matrix. Furthermore, the absolute value of each entry of A is at most $R = n^2$. Thus, by Theorem 3 we can compute $rank(A)$ in the stated running time. □

It is important to notice that computing $rank(A)$ directly in Theorem 1, without computing AA^T, is costly. The fastest algorithms for computing the rank of an $s \times n$ matrix directly require the use of Gaussian elimination, and can be performed using $O(ns^{\omega-1})$ algebraic operations [3]. Gaussian elimination, however,

cannot make use of the fact that the matrix is sparse (namely, in our terms, make use of m as a parameter to its running time). This can be attributed to the negative result of Yannakakis [20] who proved that controlling the number of fill-ins (entries that were originally zero and become non-zero during the elimination process) is an NP-Hard problem.

4 All Pairs Shortest Paths in Graphs with an s-Vertex Cover

In this section we prove Theorem 2. Suppose $G = (V, E)$ is an undirected graph and $S \subset V$ is a vertex cover of G. The weight of each edge is an integer from $\{1, \ldots, W\}$. We denote $S = \{1, \ldots, s\}$ and $T = V \setminus S = \{s+1, \ldots, n\}$. Our goal is to obtain the distance matrix $D = D_{n \times n}$ whose rows and columns are indexed by V, where $D(x, y)$ is the length of a shortest path connecting x and y.

For an $n^\alpha \times n^\beta$ matrix A and an $n^\beta \times n^\gamma$ matrix B, both with entries in $\{0, \ldots, K\} \cup \{\infty\}$, we define the *distance product* $C = A \star B$ to be $C(i, j) = Min_{k=1}^{n^\beta} A(i, k) + B(k, j)$. Yuval [22] observed that C can be computed in time $O(Kn^{\omega(\alpha,\beta,\gamma)})$. The idea of his proof is to replace each entry z with $(n^\beta + 1)^z$ (and infinity with 0), compute the usual product C' of the resulting matrices A' and B' , and deduce $C(i, j)$ by considering $C'(i, j)$ as a number written in base $n^\beta + 1$. In fact, this argument can be stated more generally as follows:

Lemma 10. *For an $n^\alpha \times n^\beta$ matrix A and an $n^\beta \times n^\gamma$ matrix B, both with entries in $\{0, \ldots, K\} \cup \{\infty\}$, let $c(i, j, q)$ denote the number of distinct indices k for which $A(i, k) + B(k, j) = q$, Then, all the numbers $c(i, j, q)$ for $i = 1, \ldots, n^\alpha$, $j = 1, \ldots, n^\gamma$ and $q = 0, \ldots, 2K$ can be computed in $\tilde{O}(Kn^\omega(\alpha, \beta, \gamma))$ time.*

Denote by A the adjacency matrix of G where the rows are indexed by S and the columns by V. Thus, A is an $s \times n$ matrix and $A(i, j) = w(i, j)$ if $ij \in E$, $A(i, i) = 0$, and otherwise $A(i, j) = \infty$, for $i = 1, \ldots, s$ and $j = 1, \ldots, n$. We first compute the distance product $B = A \star A^T$. This can be done in $\tilde{O}(Wn^{\omega(\mu,1,\mu)})$ time where $s = n^\mu$, using Lemma 10. We consider B as the adjacency matrix of a weighted undirected graph G' whose vertex set is S. Notice that the weight of each edge of G' is between 1 and $2W$. The weight $w'(i, j)$ corresponds to a shortest path connecting i with j in G, among all paths with at most two edges.

We solve the APSP problem in G'. This can be done in $\tilde{O}(Ws^\omega)$ time using the algorithm of Shoshan and Zwick [17]. Denote the output distance matrix by D'. Clearly, $D'(i, j) = D(i, j)$ for $i = 1, \ldots, s$ and $j = 1, \ldots, s$. Indeed, in a shortest path from i to j in G we can short-circuit any two consecutive edges (i_1, i_2, i_3) where $i_2 \in T$ with the direct edge (i_1, i_3) which is an edge of G', without increasing the length of the path.

We now remain with the problem of computing $D(i, j)$ where at least one of i or j is in T. By symmetry we shall assume that $i < j$. Consider first the case $i \in S$ and $j \in T$. In the beginning of the algorithm we can choose (in linear time), for each $j \in T$, an arbitrary neighbor $f(j) \in S$. We have already computed

$D(i, f(j))$; hence suppose that $D(i, f(j)) = \ell$. As our graph is undirected we have that if $\ell = \infty$ then also $D(i, j) = \infty$. Otherwise, for each neighbor v of j,

$$|D(i, v) - \ell| \leq 2W.$$

Let x_h denote the number of neighbors v of j with $D(i, v)+w(v, j) = \ell-2W+h$, for $h = 1, \ldots, 5W$. Clearly, if we can determine all the x_h then, if h' is the smallest index for which $x_{h'} > 0$ then $D(i, j) = \ell - 2W + h$.

We propose two methods for computing the x_h's. The first method is suitable (in our setting) when W is constant, but is presented here since it is also applicable in situations where the set of possible distances is *not consecutive* and not necessarily constant, thus we believe it may find other applications. Let $D^{(k)}$ be the matrix obtained from D' by replacing each entry z with z^k. We shall demonstrate the method in case $W = 1$ (the unweighted case). The generalization is not difficult. Consider the regular integer products $C_k = A^T D^{(k)}$ for $k = 0, \ldots, 4$ (for the sake of accuracy, we replace infinities with zeros when performing the integer product). Thus, $C_0(j, i)$ is just the number of neighbors of j that can reach i. Namely, $C_0(j, i) = x_1 + x_2 + x_3 + x_4 + x_5$. Similarly, $C_k(j, i)$ is just the sum of the distances from each of these neighbors to i, each distance taken to the k'th power. Namely,

$$C_k(j, i) = (\ell - 2)^k x_1 + (\ell - 1)^k x_2 + \ell^k x_3 + (\ell + 1)^k x_4 + (\ell + 2)^k x_5.$$

Considering the x_1, x_2, x_3, x_4, x_5 as unknown variables, we have a system of five linear equations whose coefficient matrix is just the 5×5 Vandermonde matrix with generators $\ell - 2, \ell - 1, \ell, \ell + 1, \ell + 2$. It is well known that this system has a unique solution (all of our generators are distinct, and even consecutive). This Vandermonde method becomes more inefficient if W is not constant, but each linear system can be solved in $O(W^2)$ time.

The second method uses truncated distance matrices. Let $\hat{D}'$ be the matrix obtained from D' by replacing each entry z with $z \bmod 5W$. We apply Lemma 10 to the product $A^T \star \hat{D}'$, and obtain, in particular, the values $c(i, j, q)$ for $q = 0, \ldots, 6W - 1$. Notice that each finite entry in A is between 0 and W and each finite entry in $\hat{D}'$ is between 0 and $5W - 1$. We claim that we can determine the x_h from these values. Indeed, each contribution to $c(i, j, q)$ is due to a neighbor v so that $(D(i, v) \bmod 5W) + w(v, j) = q$. But this determines $D(i, v) + w(v, j)$, and hence a corresponding x_h, uniquely, since we know that all the possible $D(i, v)$ are between $\ell - 2W$ and $\ell + 2W$, thus, in a $4W$ interval, and we know that $1 \leq w(v, j) \leq W$, which is another interval of length W.

We have thus shown how to compute all of the distances $D(i, j)$ for $i \in S$ and $j \in T$ precisely. In particular, we have determined the first s rows of D in $\tilde{O}(Wn^{\omega(1,\mu,\mu)})$ time. Denote these first s rows of D by D''. To determine the distances $D(i, j)$ where $i \in T$ and $j \in T$, we simply repeat the above procedure using the truncated distance matrix $\hat{D}''$ (instead of $\hat{D}'$) and applying Lemma 10 to the product $A^T \star \hat{D}''$. The running time now is $\tilde{O}(Wn^{\omega(1,\mu,1)})$.

We have shown how to correctly compute the final distance matrix D. What remains is to determine the running time of the algorithm. As noted in the

introduction, if $vc(G) = n^{\mu}$ then finding a vertex cover S with $s \leq 2n^{\mu}$ vertices can be easily done in $O(n^2)$ time. Overall, the algorithm consists of three distance products (three applications of Lemma 10), and one application of the algorithm of Shoshan and Zwick. The most time consuming operation is $\tilde{O}(Wn^{\omega(1,\mu,1)})$, and hence the result follows. □

5 Concluding Remarks

We presented two new algorithms for two fundamental problems in algorithmic combinatorics. Both of these algorithms are based on rectangular matrix multiplication, but each is combined with different tools from combinatorics and linear algebra. It is plausible that for Maximum Subset Matching this is not the end of the road. The possibility of a faster algorithm remains (whether using fast matrix multiplication or not). This, however, is not the case for our APSP result. Namely, no algorithm that is expressed in terms of n and $vc(G)$ can outperform the algorithm of Theorem 2. Assuming $vc(G) = s$, no algorithm can perform faster than the time needed to multiply two Boolean matrices of orders $n \times s$ and $s \times n$.

To see this, consider the following simple reduction. Let A be an $n \times s$ Boolean matrix and let B be an $s \times n$ Boolean matrix. Create a graph with $2n+s$ vertices, consisting of sets $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_s\}$, $Z = \{z_1, \ldots, z_n\}$. There is an edge from $x_i \in X$ to $y_j \in Y$ if an only if $A(i, j) = 1$. Similarly, there is an edge from $y_i \in Y$ to $z_j \in Z$ if an only if $B(i, j) = 1$. Notice that Y is a vertex cover of the created graph. Clearly, the shortest path from x_i to z_j has length 2 if and only if in the product $C = AB$ we have $C(i, j) = 1$.

Another interesting remark, pointed out by one of the referees of this paper, is that Theorem 3 can be extended to finite fields. To do so note, first that by the Cauchy Binet formula, if A has full row-rank, call it r, and D is a random diagonal matrix with independent variables in its diagonal, then the determinant of ADA^T is a nonzero polynomial of degree r in these variables. This implies that in general for D as above, the rank of ADA^T is equal to that of A. If the field is large enough (much larger than the rank), then by the results of Zippel [23] and Schwartz [15] this implies that substituting random elements of the field for the diagonal entries of D would give, with high probability, a matrix with the same rank as that of A. Otherwise, we can substitute random elements in a sufficiently large extension field, noting that this involves only an extra logarithmic factor in the complexity.

References

1. Alon, N., Naor, M.: Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. Algorithmica 16, 434–449 (1996)
2. Blum, N.: A New Approach to Maximum Matching in General Graphs. In: Paterson, M.S. (ed.) Automata, Languages and Programming. LNCS, vol. 443, pp. 586–597. Springer, Heidelberg (1990)

3. Bunch, J., Hopcroft, J.: Triangular factorization and inversion by fast matrix multiplication. Mathematics of Computation 28, 231–236 (1974)
4. Coppersmith, D.: Rectangular matrix multiplication revisited. Journal of Complexity 13, 42–49 (1997)
5. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation 9, 251–280 (1990)
6. Edmonds, J.: Paths, trees, and flowers. Canadian Journal of Mathematics 17, 449–467 (1965)
7. Gabow, H.N., Tarjan, R.E.: Faster scaling algorithms for general graph matching problems. Journal of the ACM 38(4), 815–853 (1991)
8. Galil, Z., Margalit, O.: All Pairs Shortest Paths for graphs with small integer length edges. Journal of Computer and System Sciences 54, 243–254 (1997)
9. Harvey, N.: Algebraic Structures and Algorithms for Matching and Matroid Problems. Proceedings of the 47$^{\text{th}}$ IEEE Symposium on Foundations of Computer Science (FOCS), Berkeley, CA (October 2006)
10. Huang, X., Pan, V.Y.: Fast rectangular matrix multiplications and applications. Journal of Complexity 14, 257–299 (1998)
11. Kaplan, H., Sharir, M., Verbin, E.: Colored intersection searching via sparse rectangular matrix multiplication. In: Proceedings of the 22$^{\text{nd}}$ ACM Symposium on Computational Geometry (SOCG), pp. 52–60 (2006)
12. Lovász, L.: On determinants, matchings, and random algorithms. In: Fundamentals of computation theory, vol. 2, pp. 565–574. Akademie, Berlin (1979)
13. Micali, S., Vazirani, V.V.: An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs. In: Proceedings of the 21$^{\text{st}}$ IEEE Symposium on Foundations of Computer Science (FOCS), pp. 17–27, 1980.
14. Mucha, M., Sankowski, P.: Maximum Matchings via Gaussian Elimination. In: Proceedings of the 45$^{\text{th}}$ IEEE Symposium on Foundations of Computer Science (FOCS), pp. 248–255. IEEE Computer Society Press, Los Alamitos (2004)
15. Schwartz, J.T.: Fast Probabilistic Algorithms for Verification of Polynomial Identities. Journal of the ACM 27(4), 701–717 (1980)
16. Seidel, R.: On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs. Journal of Computer and System Sciences 51(3), 400–403 (1995)
17. Shoshan, A., Zwick, U.: All pairs shortest paths in undirected graphs with integer weights. In: Proceedings of the 40$^{\text{th}}$ Symposium of Foundations of Computer Science (FOCS), pp. 605–614 (1999)
18. Tutte, W.T.: The factorization of linear graphs. Journal of the London Mathematical Society 22, 107–111 (1947)
19. Vazirani, V.V.: A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm. Combinatorica 14(1), 71–109 (1994)
20. Yannakakis, M.: Computing the minimum fill-in is NP-complete. SIAM Journal on Algebraic and Discrete Methods 2(1), 77–79 (1981)
21. Yuster, R., Zwick, U.: Fast sparse matrix multiplication. ACM Transactions on Algorithms 1, 2–13 (2005)
22. Yuval, G.: An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision multiplications. Information Processing Letters 4, 155–156 (1976)
23. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: Ng, K.W. (ed.) Symbolic and Algebraic Computation. LNCS, vol. 72, pp. 216–226. Springer, Heidelberg (1979)

Linear-Time Ranking of Permutations

Martin Mareš* and Milan Straka

Department of Applied Mathematics and
Institute of Theoretical Computer Science (ITI)
Charles University
Malostranské nám. 25, 118 00 Praha, Czech Republic
{mares,fox}@kam.mff.cuni.cz

Abstract. A *lexicographic ranking function* for the set of all permutations of n ordered symbols translates permutations to their ranks in the lexicographic order of all permutations. This is frequently used for indexing data structures by permutations. We present algorithms for computing both the ranking function and its inverse using $O(n)$ arithmetic operations.

1 Introduction

A permutation of order n is a bijection of an n-element set X onto itself. For convenience, we will always assume that $X = [n] = \{1, \ldots, n\}$, so the permutations become ordered n-tuples containing each number $1, \ldots, n$ exactly once.

Many applications ask for arrays indexed by permutations. Among other uses documented in [1], this is handy when searching for Hamilton cycles in Cayley graphs [2,3] or when exploring state spaces of combinatorial puzzles like the Loyd's Fifteen [4]. To accomplish that, a *ranking function* is usually employed, which translates a permutation π to a unique number $R(\pi) \in \{0, \ldots, n! - 1\}$. The inverse of the ranking function $R^{-1}(i)$ is also frequently used and it is called the *unranking function.*

Each ranking function corresponds to a linear order on the set of all permutations (it returns the number of permutations which are strictly less than the given one). The traditional approach to the ranking problem is to fix lexicographic order and construct the appropriate ranking and unranking functions. In fact, an arbitrary order suffices in many cases, but the lexicographic ranking has the additional advantage of a nice structure allowing additional operations on permutations to be performed directly on their ranks.

Naïve implementations of lexicographic ranking require time $\Theta(n^2)$ in the worst case [5,6]. This can be easily improved to $O(n \log n)$ by using either a binary search tree to calculate inversions, or by a divide-and-conquer technique or by clever use of modular arithmetic (all three algorithms are described in [7]). Myrvold and Ruskey [8] mention further improvements to $O(n \log n / \log \log n)$ by using the data structures of Dietz [9].

* Supported by grant 1M0021620808 of the Czech Ministry of Education.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 187–193, 2007.

Linear time complexity was reached also by Myrvold and Ruskey [8] by employing a different order, which is defined locally by the history of the data structure — in fact, they introduce a linear-time unranking algorithm first and then they derive an inverse algorithm without describing the order explicitly. However, they leave the problem of lexicographic ranking open.

In this paper, we present a linear-time algorithm for the lexicographic order. It is based on an observation that once we assume that our computation model is capable of performing basic arithmetics on numbers of the order of magnitude of the resulting rank, we can use such integers to represent fairly rich data structures working in constant time. This approach has been pioneered by Fredman and Willard [10,11], who presented Fusion trees and Atomic heaps working in various RAM models. Our structures are built on similar principles, but in a simpler setting, therefore we even can avoid random access arrays and our algorithms work in almost all models of computation, relying only on the usual set of arithmetic and logical operations on the appropriately large integers.

We also extend our algorithm to ranking and unranking of k-permutations, i.e., ordered k-tuples of distinct elements drawn from $[n]$.

2 Ranking Permutations

Permutations have a simple recursive structure: if we fix the first element $\pi[1]$ of a permutation π on $[n] = \{1, \ldots, n\}$, the elements $\pi[2], \ldots, \pi[n]$ form a permutation on $\{1, \ldots, \pi[1]-1, \pi[1]+1, \ldots, n\}$. The lexicographic order of π and π' is then determined by $\pi[1]$ and $\pi'[1]$ and only if these elements are equal, it is decided by the lexicographic comparison of permutations $(\pi[2], \ldots, \pi[n])$ and $(\pi'[2], \ldots, \pi'[n])$. Therefore the rank of π is $(\pi[1]-1) \cdot (n-1)!$ plus the rank of $(\pi[2], \ldots, \pi[n])$.

This gives a reduction of (un)ranking of permutations on $[n]$ to (un)ranking of permutations on a $(n-1)$-element set, which suggests a straightforward algorithm, but unfortunately this set is different from $[n-1]$ and it even depends on the value of $\pi[1]$. We could renumber the elements to get $[n-1]$, but it would require linear time per iteration. Instead, we generalize the problem to permutations on subsets of $[n]$. Therefore for a permutation π on $A \subseteq [n]$ we have:

$$R((\pi[1], \ldots, \pi[m]), A) = r(\pi[1], A) \cdot (m-1)! + R((\pi[2], \ldots, \pi[m]), A \setminus \{\pi[1]\}),$$

where $r(x, A)$ is a ranking function on elements of A.

This recurrence leads to the following well known algorithms (see for example [7]) for ranking and unranking. Here $\pi[i, \ldots, n]$ denotes the array containing the permutation and A is a data structure representing the subset of $[n]$ on which π is defined, i.e., $A = \{\pi(j) : i \leq j \leq n\}$. This structure supports ranking, unranking and deletion of individual elements. We will denote these operations $r(x, A)$, $r^{-1}(i, A)$ and $A \setminus \{x\}$ respectively.

function rank(π, i, n, A) { return the rank of perm. $\pi[1, \ldots, n]$ on A }
  if $i \geq n$ **then return** 0
  $a \leftarrow r(\pi[i], A)$
  $b \leftarrow \text{rank}(\pi, i+1, n, A \setminus \{\pi[i]\})$
  return $a \cdot (n-i)! + b$
end

function rank(π, n) { return the rank of a permutation π on $[n]$ }
  return rank$(\pi, 1, n, [n])$
end

function unrank(j, i, n, A) { get $\pi[i, \ldots, n]$ of the j-th perm. π on A }
  if $i > n$ **then return** $(0, \ldots, 0)$
  $a \leftarrow r^{-1}(\lfloor j/(n-i)! \rfloor, A)$
  $\pi \leftarrow \text{unrank}(j \bmod (n-i)!, i+1, n, A \setminus \{\pi[i]\})$
  $\pi[i] \leftarrow a$
end

function unrank(j, n) { return the j-th permutation on $[n]$ }
  return unrank$(j, 1, n, [n])$
end

If we precalculate the factorials and assume that each operation on the data structure A takes time at most $t(n)$, both algorithms run in time $O(n \cdot t(n))$ and their correctness follows from the discussion above.

A trivial implementation of the data structure by an array yields $t(n) = O(n)$. Using a binary search tree instead gives $t(n) = O(\log n)$. The data structure of Dietz [9] improves it to $t(n) = O(\log n / \log \log n)$. In fact, all these variants are equivalent to the classical algorithms based on inversion vectors, because at the time of processing $\pi[i]$, the value of $r(\pi[i], A)$ is exactly the number of elements forming inversions with $\pi[i]$.

If we relax the requirements on the data structure to allow ordering of elements dependent on the history of the structure (i.e., on the sequence of deletes performed so far), we can implement all three operations in time $O(1)$. We store the values in an array and we also keep an inverse permutation. Ranking is done by direct indexing, unranking by indexing of the inverse permutation and deleting by swapping the element with the last element. We can observe that although it no longer gives the lexicographic order, the unranking function is still the inverse of the ranking function (the sequence of deletes from A is the same in both functions), so this leads to $O(n)$ time ranking and unranking in a non-lexicographic order. This order is the same as the one used by Myrvold and Ruskey in [8].

However, for our purposes we need to keep the same order on A over the whole course of the algorithm.

3 Word-Encoded Sets

We will describe a data structure for the subsets of $[n]$ supporting insertion, deletion, ranking and unranking in constant time per operation in the worst case. It uses similar techniques as the structures introduced by Fredman and Willard in [10,11], but our setting allows for a much simpler implementation.

First, let us observe that whatever our computation model is, it must allow operations with integers up to $n!-1$, because the ranks of permutations can reach such large numbers. In accordance with common practice, we will assume that the usual set of arithmetic and logical operations is available and that they work in constant time on integers of that size, i.e., on $\lceil \log_2(n!) \rceil = \Omega(n \log n)$ bits. Furthermore, multiple-precision arithmetics on numbers which fit in a constant number of machine words can be emulated with a constant number of operations on these words, so we can also assume existence of constant-time operations on arbitrary $O(n \log n)$-bit numbers. This gives us an opportunity for using the integers to encode data structures.

Let us denote $b = \lceil \log_2(n+1) \rceil$ the number of bits needed to represent a single element of $[n]$. We will store the whole subset $A \subseteq [n]$, $\ell = |A|$, as a bit vector $\mathbf{a}$ consisting of ℓ fields of size b, each field containing a single element of A. The fields will be maintained in increasing order of values and an additional zero bit will be kept between adjacent fields and also above the highest field. The vector is $\ell(b+1) = O(n \log n)$ bits long, so it fits in $O(1)$ integers. We will describe how to perform the data structure operations using arithmetic and logical operations on these integers.

Unranking is just extraction of the r-th field of the vector. It can be accomplished by shifting the representation of $\mathbf{a}$ by $r \cdot (b+1)$ bits to the right and ANDing it with $2^b - 1$, which masks out the high-order bits.

Insertion can be reduced to ranking: once we know the position in the vector the new element should land at, i.e., its rank, we can employ bit shifts, ANDs and ORs to shift apart the existing fields and put the new element to the right place.

Deletion can be done in a similar way. The rank gives us the position of the field we want to delete and we again use bit operations to move the other fields in parallel.

Ranking of an element x is the only non-trivial operation. We prepare a vector $\mathbf{c}$, which has all ℓ fields set to x (this can be done in a single multiplication by an appropriate constant) and the separator bits between them set to ones. Observe that if we subtract $\mathbf{a}$ from $\mathbf{c}$, the separator bits change to zeroes exactly at the places where $a[i] > c[i] = x$ (and they absorb the carries, so the fields do not interfere with each other). Therefore the desired rank is the number of remaining ones in the separator bits minus 1.

Let us observe that if $\mathbf{z}$ is an encoding of a vector with separator zeroes, then $\mathbf{z} \bmod (2^{b+1} - 1)$ is the sum of all fields of the vector modulo $2^{b+1} - 1$. This works because $\mathbf{z} = \sum_i z[i] \cdot 2^{(b+1)i}$ and $2^{(b+1)i} \bmod (2^{b+1} - 1) = 1$ for every i. Hence the separator bits in $\mathbf{c}$ can be summed by masking out all non-separator bits, shifting $\mathbf{c}$ to the right to transform the separator bits to field values and calculating $\mathbf{c} \bmod (2^{b+1} - 1)$.

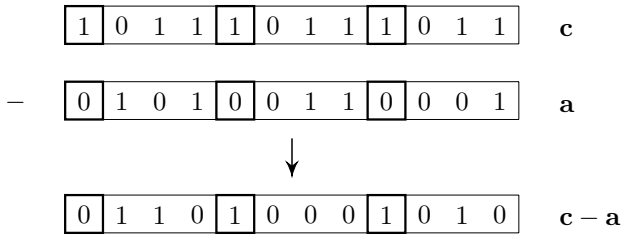


Fig. 1. Computing the rank of element 3 in a set $A = \{1, 3, 5\}$. The bits framed in bold are the separator bits, **c** is the vector containing copies of the number 3.

All four operations work in constant time. The auxiliary constants for bit patterns we needed can be precalculated in linear time at the start of the algorithm (we could even calculate them in constant time, as $2^0 + 2^q + 2^{2q} + \ldots + 2^{pq} = (2^{(p+1)q} - 1)/(2^q - 1)$, but it is an unnecessary complication). With this data structure, the ranking and unranking algorithms achieve linear time complexity.

4 Ranking *k*-Permutations

Our (un)ranking algorithms can be used for k-permutations as well if we just stop earlier and divide everything by $(n - k)!$. Unfortunately, the ranks of k-permutations can be much smaller, so we can no longer rely on the data structure fitting in a constant number of integers. For example, if $k = 1$, the ranks are $O(\log n)$-bit numbers, but the data structure still requires $\Theta(n \log n)$ bits.

We do a minor side step by remembering the complement of A instead, that is the set of the at most k elements we have already seen. We will call it G (and the corresponding vector **g**), because they are the gaps in A. Let us prove that $\Omega(k \log n)$ bits are needed to store the rank, which is enough space to represent the whole **g**.

Lemma 1. *The number of k-permutations on $[n]$ is $2^{\Omega(k \log n)}$.*

Proof. There are $n^{\underline{k}} = n(n-1)\ldots(n-k+1)$ such k-permutations. If $k \leq n/2$, then every term in the product is at least $n/2$, so $\log_2 n^{\underline{k}} \geq k(\log_2 n - 1)$. If $k \geq n/2$, then $n^{\underline{k}} \geq n^{\underline{n/2}}$ and $\log_2 n^{\underline{k}} \geq (n/2)(\log_2 n - 1) \geq (k/2)(\log_2 n - 1)$. □

Deletes in A now become inserts in G. The rank of x in A is just x minus the rank of the largest element of G which is smaller than x. This is easy to get, since when we ask our data structure for a rank of an element x outside the set, we get exactly the number elements of the set smaller than x.

The only operation we cannot translate directly is unranking in A. To achieve that, we will maintain another vector **b** such that $b[i] = g[i] - i$, which is the number of elements in A smaller than $g[i]$. Now, if we want to find the r-th element of A, we find the largest i such that $b[i] < r$ (the rank of r in **b** in the

same sense as above) and we return $g[i] + 1 + r - b[i]$. This works because there is no gap in A between element $g[i] + 1$, which has rank $b[i]$, and the desired element of rank r.

For example, if $A = \{2, 5, 6\}$ and $n = 8$, then $\mathbf{g} = (1, 3, 4, 7, 8)$ and $\mathbf{b} = (0, 1, 1, 3, 3)$. When we want to calculate $r^{-1}(2, A)$, we find $i = 3$ and we get $g[3] + 1 + 2 - b[3] = 4 + 1 + 2 - 1 = 6$.

Also, whenever a new element is inserted into $\mathbf{g}$, we can shift the fields of $\mathbf{b}$ accordingly and decrease all higher fields by one in parallel by a single subtraction.

We have replaced all operations on A by the corresponding operations on the modified data structure, each of which works again in constant time. This gives us a linear-time algorithm for the k-permutations, too.

5 Concluding Remarks

We have shown linear-time algorithms for ranking and unranking of permutations and k-permutations on integers, closing a frequently encountered open question.

Our algorithms work in linear time in a fairly broad set of computation models. Even if we take the complexity of operations on numbers in account, we have proven that the number of arithmetic operations for determining the inversion vector is bounded by the number of those required to calculate the rank from the inversion vector, contrary to previous intuition.

The technique we have demonstrated should give efficient algorithms for ranking of various other classes of combinatorial objects, if the number of such objects is high enough to ensure word size suitable for our data structures.

References

1. Critani, F., Dall'Aglio, M., Di Biase, G.: Ranking and unranking permutations with applications. In: Innovation in Mathematics. In: Proceedings of Second International Mathematica Symposium, pp. 99–106 (1997)
2. Ruskey, F., Jiang, M., Weston, A.: The Hamiltonicity of directed-Cayley graphs (or: A tale of backtracking). Discrete Appl. Math. 57, 75–83 (1995)
3. Ruskey, F., Savage, C.: Hamilton Cycles that Extend Transposition Matchings in Cayley Graphs of Sn. SIAM Journal on Discrete Mathematics 6(1), 152–166 (1993)
4. Slocum, J., Sonneveld, D.: The 15 Puzzle Book. The Slocum Puzzle Foundation, Beverly Hills, CA, USA (2006)
5. Liebehenschel, J.: Ranking and Unranking of Lexicographically Ordered Words: An Average-Case Analysis. Journal of Automata, Languages and Combinatorics 2(4), 227–268 (1997)
6. Reingold, E.: Combinatorial Algorithms: Theory and Practice. Prentice Hall College Div., Englewood Cliffs (1977)
7. Knuth, D.: The Art of Computer Programming, Sorting and Searching, vol. 3. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA (1998)
8. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters 79(6), 281–284 (2001)

9. Dietz, P.F.: Optimal algorithms for list indexing and subset rank. In: Dehne, F., Santoro, N., Sack, J.-R. (eds.) WADS 1989. LNCS, vol. 382, pp. 39–46. Springer, Heidelberg (1989)
10. Fredman, M., Willard, D.: Surpassing the information theoretic bound with fusion trees. Journal of Computer and System Sciences 47(3), 424–436 (1993)
11. Fredman, M., Willard, D.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. Journal of Computer and System Sciences 48(3), 533–551 (1994)

Radix Sorting with No Extra Space*

Gianni Franceschini[1], S. Muthukrishnan[2], and Mihai Pătraşcu[3]

[1] Dept of Computer Science, Univ. of Pisa
francesc@di.unipi.it
[2] Google Inc., NY
muthu@google.com
[3] MIT, Boston
mip@mit.edu

Abstract. It is well known that n integers in the range $[1, n^c]$ can be sorted in $O(n)$ time in the RAM model using radix sorting. More generally, integers in any range $[1, U]$ can be sorted in $O(n\sqrt{\log \log n})$ time [5]. However, these algorithms use $O(n)$ words of extra memory. Is this necessary?

We present a simple, stable, integer sorting algorithm for words of size $O(\log n)$, which works in $O(n)$ time and uses only $O(1)$ words of extra memory on a RAM model. This is the integer sorting case most useful in practice. We extend this result with same bounds to the case when the keys are read-only, which is of theoretical interest. Another interesting question is the case of arbitrary c. Here we present a black-box transformation from any RAM sorting algorithm to a sorting algorithm which uses only $O(1)$ extra space and has the same running time. This settles the complexity of in-place sorting in terms of the complexity of sorting.

1 Introduction

Given n integer *keys* $S[1 \dots n]$ each in the range $[1, n]$, they can be sorted in $O(n)$ time using $O(n)$ space by *bucket sorting*. This can be extended to the case when the keys are in the range $[1, n^c]$ for some positive constant c by *radix sorting* that uses repeated bucket sorting with $O(n)$ ranged keys. The crucial point is to do each bucket sorting *stably*, that is, if positions i and j, $i < j$, had the same key k, then the copy of k from position i appears before that from position j in the final sorted order. Radix sorting takes $O(cn)$ time and $O(n)$ space. More generally, RAM sorting with integers in the range $[1, U]$ is a much-studied problem. Currently, the best known bound is the randomized algorithm in [5] that takes $O(n\sqrt{\log \log n})$ time, and the deterministic algorithm in [1] that takes $O(n \log \log n)$ time. These algorithms also use $O(n)$ words of extra memory in addition to the input.

We ask a basic question: *do we need $O(n)$ auxiliary space for integer sorting*? The ultimate goal would be to design *in-place* algorithms for integer sorting that uses only $O(1)$ extra words, This question has been explored in depth for comparison-based sorting, and after a series of papers, we now know that in-place, stable comparison-based sorting can be done in $O(n \log n)$ time [10]. Some very nice algorithmic techniques have been developed in this quest. However, no such results

* A full version of this paper is available as arXiv:0706.4107.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 194–205, 2007.

are known for the integer sorting case. Integer sorting is used as a subroutine in a number of algorithms that deal with trees and graphs, including, in particular, sorting the transitions of a finite state machine. Indeed, the problem arose in that context for us. In these applications, it is useful if one can sort in-place in $O(n)$ time. From a theoretical perspective, it is likewise interesting to know if the progress in RAM sorting, including [3,1,5], really needs extra space.

Our results are in-place algorithms for integer sorting. Taken together, these results solve much of the issues with space efficiency of integer sorting problems. In particular, our contributions are threefold.

A practical algorithm. In Section 2, we present a stable integer sorting algorithm for $O(\log n)$ sized words that takes $O(n)$ time and uses only $O(1)$ extra words.

This algorithm is a simple and practical replacement to radix sort. In the numerous applications where radix sorting is used, this algorithm can be used to improve the space usage from $O(n)$ to only $O(1)$ extra words. We have implemented the algorithm with positive results.

One key idea of the algorithm is to compress a portion of the input, modifying the keys. The space thus made free is used as extra space for sorting the remainder of the input.

Read-only keys. It is theoretically interesting if integer sorting can be performed in-place *without modifying the keys.* The algorithm above does not satisfy this constraint. In Section 3, we present a more sophisticated algorithm that still takes linear time and uses only $O(1)$ extra words without modifying the keys. In contrast to the previous algorithm, we cannot create space for ourselves by compressing keys. Instead, we introduce a new technique of *pseudo pointers* which we believe will find applications in other succinct data structure problems. The technique is based on keeping a set of distinct keys as a pool of preset read-only pointers in order to maintain linked lists as in bucket sorting.

As a theoretical exercise, the full version of this paper also considers the case when this sorting has to be done stably. We present an algorithm with identical performance that is also stable. Similar to the other in-place stable sorting algorithms e.g., comparison-based sorting [10], this algorithm is quite detailed and needs very careful management of keys as they are permuted. The resulting algorithm is likely not of practical value, but it is still fundamentally important to know that bucket and radix sorting can indeed be solved stably in $O(n)$ time with only $O(1)$ words of extra space. For example, even though comparison-based sorting has been well studied at least since 60's, it was not until much later that optimal, stable in-place comparison-based sorting was developed [10].

Arbitrary word length. Another question of fundamental theoretical interest is whether the recently discovered integer sorting algorithms that work with long keys and sort in $o(n \log n)$ time, such as [3,1,5], need any auxiliary space. In Section 4, we present a black-box transformation from any RAM sorting algorithm to an sorting algorithm which uses only $O(1)$ extra space, and retains the same time bounds. As a result, the running time bounds of [1,5] can now be matched with only $O(1)$ extra space. This transformation relies on a fairly natural technique of compressing a portion of the input to make space for simulating space-inefficient RAM sorting algorithms.

Definitions. Formally, we are given a sequence S of n elements. The problem is to sort S according to the integer keys, under the following assumptions:

(i) Each element has an integer key within the interval $[1, U]$.
(ii) The following unit-cost operations are allowed on S: (a) indirect address of any position of S; (b) read-only access to the key of any element; (c) exchange of the positions of any two elements.
(iii) The following unit-cost operations are allowed on integer values of $O(\log U)$ bits: addition, subtraction, bitwise AND/OR and unrestricted bit shift.
(iv) Only $O(1)$ auxiliary words of memory are allowed; each word had $\log U$ bits.

For the sake of presentation, we will refer to the elements' keys as if they were the input elements. For example, for any two elements x, y, instead of writing that the key of x is less than the key of y we will simply write $x < y$. We also need a precise definition of the *rank* of an element in a sequence when multiple occurrences of keys are allowed: the *rank* of an element x_i in a sequence $x_1 \dots x_t$ is the cardinality of the multiset $\{x_j \mid x_j < x_i \text{ or } (x_j = x_i \text{ and } j \leq i)\}$.

2 Stable Sorting for Modifiable Keys

We now describe our simple algorithm for (stable) radix sort without additional memory.

Gaining space. The first observation is that numbers in sorted order have less entropy than in arbitrary order. In particular, n numbers from a universe of u have binary entropy $n \log u$ when the order is unspecified, but only $\log \binom{u}{n} = n \log u - \Theta(n \log n)$ in sorted order. This suggests that we can "compress" sorted numbers to gain more space:

Lemma 1. *A list of n integers in sorted order can be represented as: (a) an array $A[1 \dots n]$ with the integers in order; (b) an array of n integers, such that the last $\Theta(n \log n)$ bits of the array are zero. Furthermore, there exist in-place $O(n)$ time algorithms for switching between representations (a) and (b).*

Proof. One can imagine many representations (b) for which the lemma is true. We note nonetheless that some care is needed, as some obvious representations will in fact not lead to in-place encoding. Take for instance the appealing approach of replacing $A[i]$ by $A[i] - S[i-1]$, which makes numbers tend to be small (the average value is $\frac{u}{n}$). Then, one can try to encode the difference using a code optimized for smaller integers, for example one that represents a value x using $\log x + O(\log \log x)$ bits. However, the obvious encoding algorithm will not be in-place: even though the scheme is guaranteed to save space over the entire array, it is possible for many large values to cluster at the beginning, leading to a rather large prefix being in fact expanded. This makes it hard to construct the encoding in the same space as the original numbers, since we need to shift a lot of data to the right before we start seeing a space saving.

As it will turn out, the practical performance of our radix sort is rather insensitive to the exact space saving achieved here. Thus, we aim for a representation

which makes in-place encoding particularly easy to implement, sacrificing constant factors in the space saving.

First consider the most significant bit of all integers. Observe that if we only remember the minimum i such that $A[i] \geq u/2$, we know all most significant bits (they are zero up to i and one after that). We will encode the last $n/3$ values in the array more compactly, and use the most significant bits of $A[1 \ldots \frac{2}{3}n]$ to store a stream of $\frac{2}{3}n$ bits needed by the encoding.

We now break a number x into $\mathrm{hi}(x)$, containing the upper $\lfloor \log_2(n/3) \rfloor$ bits, and $\mathrm{lo}(x)$, with the low $\log u - \lfloor \log_2(n/3) \rfloor$ bits. For all values in $A[\frac{2}{3}n+1 \ldots n]$, we can throw away $\mathrm{hi}(A[i])$ as follows. First we add $\mathrm{hi}(A[\frac{2}{3}n+1])$ zeros to the bit stream, followed by a one; then for every $i = \frac{2}{3}n+2, \ldots, n$ we add $\mathrm{hi}(A[i]) - \mathrm{hi}(A[i-1])$ zeros, followed by a one. In total, the stream contains exactly $n/3$ ones (one per element), and exactly $\mathrm{hi}(A[n]) \leq n/3$ zeros. Now we simply compact $\mathrm{lo}(A[\frac{2}{3}n+1]), \ldots, \mathrm{lo}(A[n])$ in one pass, gaining $\frac{n}{3}\lfloor \log_2(n/3) \rfloor$ free bits. □

An unstable algorithm. Even just this compression observation is enough to give a simple algorithm, whose only disadvantage is that it is unstable. The algorithm has the following structure:

1. sort the subsequence $S[1 \ldots (n/\log n)]$ using the optimal in-place mergesort in [10].
2. compress $S[1 \ldots (n/\log n)]$ by Lemma 1, generating $\Omega(n)$ bits of free space.
3. radix sort $S[(n/\log n)+1 \ldots n]$ using the free space.
4. uncompress $S[1 \ldots (n/\log n)]$.
5. merge the two sorted sequences $S[1 \ldots (n/\log n)]$ and $S[(n/\log n)+1 \ldots n]$ by using the in-place, linear time merge in [10].

The only problematic step is 3. The implementation of this step is based on the cycle leader permuting approach where a sequence A is re-arranged by following the cycles of a permutation π. First $A[1]$ is sent in its final position $\pi(1)$. Then, the element that was in $\pi(1)$ is sent to its final position $\pi(\pi(1))$. The process proceeds in this way until the cycle is closed, that is until the element that is moved in position 1 is found. At this point, the elements starting from $A[2]$ are scanned until a new cycle leader $A[i]$ (i.e. its cycle has not been walked through) is found, $A[i]$'s cycle is followed in its turn, and so forth.

To sort, we use $2n^\epsilon$ counters $c_1, \ldots, c_{n^\epsilon}$ and $d_1, \ldots, d_{n^\epsilon}$. They are stored in the auxiliary words obtained in step 2. Each d_j is initialized to 0. With a first scan of the elements, we store in any c_i the number of occurrences of key i. Then, for each $i = 2 \ldots n^\epsilon$, we set $c_i = c_{i-1} + 1$ and finally we set $c_1 = 1$ (in the end, for any i we have that $c_i = \sum_{j<i} c_j + 1$). Now we have all the information for the cycle leader process. Letting $j = (n/\log n)+1$, we proceed as follows:

(*i*) let i be the key of $S[j]$;
(*ii*) if $c_i \leq j < c_{i+1}$ then $S[j]$ is already in its final position, hence we increment j by 1 and go to step (*i*);
(*iii*) otherwise, we exchange $S[j]$ with $S[c_i + d_i]$, we increment d_i by 1 and we go to step (*i*).

Note that this algorithm is inherently unstable, because we cannot differentiate elements which should fall between c_i and $c_{i+1} - 1$, given the free space we have.

Stability through recursion. To achieve stability, we need more than n free bits, which we can achieve by bootstrapping with our own sorting algorithm, instead of merge sort. There is also an important practical advantage to the new stable approach: the elements are permuted much more conservatively, resulting in better cache performance.

1. recursively sort a constant fraction of the array, say $S[1 \dots n/2]$.
2. compress $S[1 \dots n/2]$ by Lemma 1, generating $\Omega(n \log n)$ bits of free space.
3. for a small enough constant γ, break the remaining $n/2$ elements into chunks of γn numbers. Each chunk is sorted by a classic radix sort algorithm which uses the available space.
4. uncompress $S[1 \dots n/2]$.
5. we now have $1 + 1/\gamma = O(1)$ sorted subarrays. We merge them in linear time using the stable in-place algorithm of [10].

We note that the recursion can in fact be implemented bottom up, so there is no need for a stack of superconstant space. For the base case, we can use bubble sort when we are down to $n \leq \sqrt{n_0}$ elements, where n_0 is the original size of the array at the top level of the recursion.

Steps 2 and 4 are known to take $O(n)$ time. For step 3, note that radix sort in base R applied to N numbers requires $N + R$ additional words of space, and takes time $O(N \log_R u)$. Since we have a free space of $\Omega(n \log n)$ bits or $\Omega(n)$ words, we can set $N = R = \gamma n$, for a small enough constant γ. As we always have $n = \Omega(\sqrt{n_0}) = u^{\Omega(1)}$, radix sort will take linear time.

The running time is described by the recursion $T(n) = T(n/2) + O(n)$, yielding $T(n) = O(n)$.

A self-contained algorithm. Unfortunately, all algorithms so far use in-place stable merging algorithm as in [10]. We want to remove this dependence, and obtain a simple and practical sorting algorithm. By creating free space through compression at the right times, we can instead use a simple merging implementation that needs additional space. We first observe the following:

Lemma 2. *Let $k \geq 2$ and $\alpha > 0$ be arbitrary constants. Given k sorted lists of n/k elements, and αn words of free space, we can merge the lists in $O(n)$ time.*

Proof. We divide space into blocks of $\alpha n/(k+1)$ words. Initially, we have $k+1$ free blocks. We start merging the lists, writing the output in these blocks. Whenever we are out of free blocks, we look for additional blocks which have become free in the original sorted lists. In each list, the merging pointer may be inside some block, making it yet unavailable. However, we can only have k such partially consumed blocks, accounting for less than $k\frac{\alpha n}{k+1}$ wasted words of space. Since in total there are αn free words, there must always be at least one block which is available, and we can direct further output into it.

At the end, we have the merging of the lists, but the output appears in a nontrivial order of the blocks. Since there are $(k+1)(1+1/\alpha) = O(1)$ blocks in total, we can remember this order using constant additional space. Then, we can permute the blocks in linear time, obtaining the true sorted order. □

Since we need additional space for merging, we can never work with the entire array at the same time. However, we can now use a classic sorting idea, which is often used in introductory algorithms courses to illustrate recursion (see, e.g. [2]). To sort n numbers, one can first sort the first $\frac{2}{3}n$ numbers (recursively), then the last $\frac{2}{3}n$ numbers, and then the first $\frac{2}{3}n$ numbers again. Though normally this algorithm gives a running time of $\omega(n^2)$, it works efficiently in our case because we do not need recursion:

1. sort $S[1 \ldots n/3]$ recursively.
2. compress $S[1 \ldots \frac{n}{3}]$, and sort $S[\frac{n}{3}+1 \ldots n]$ as before: first radix sort chunks of γn numbers, and then merge all chunks by Lemma 2 using the available space. Finally, uncompress $S[1 \ldots \frac{n}{3}]$.
3. compress $S[\frac{2n}{3}+1 \ldots n]$, which is now sorted. Using Lemma 2, merge $S[1 \ldots \frac{n}{3}]$ with $S[\frac{n}{3}+1 \ldots \frac{2n}{3}]$. Finally uncompress.
4. once again, compress $S[1 \ldots \frac{n}{3}]$, merge $S[\frac{n}{3}+1 \ldots \frac{2n}{3}]$ with $S[\frac{2n}{3}+1 \ldots n]$, and uncompress.

Note that steps 2–4 are linear time. Then, we have the recursion $T(n) = T(n/3)+O(n)$, solving to $T(n) = O(n)$. Finally, we note that stability of the algorithm follows immediately from stability of classic radix sort and stability of merging.

Practical experience. The algorithm is surprisingly effective in practice. It can be implemented in about 150 lines of C code. Experiments with sorting 1-10 million 32-bit numbers on a Pentium machine indicate the algorithm is roughly 2.5 times slower than radix sort with additional memory, and slightly faster than quicksort (which is not even stable).

3 Unstable Sorting for Read-Only Keys

3.1 Simulating Auxiliary Bits

With the bit stealing technique [9], a bit of information is encoded in the relative order of a pair of elements with different keys: the pair is maintained in increasing order to encode a 0 and vice versa. The obvious drawback of this technique is that the cost of accessing a word of w encoded bits is $O(w)$ in the worst case (no word-level parallelism). However, if we modify an encoded word with a series of l increments (or decrements) by 1, the total cost of the entire series is $O(l)$ (see [2]).

To find pairs of distinct elements, we go from S to a sequence $Z'Y'XY''Z''$ with two properties. (i) For any $z' \in Z'$, $y' \in Y'$, $x \in X$, $y'' \in Y''$ and $z'' \in Z''$ we have that $z' < y' < x < y'' < z''$. (ii) Let $m = \alpha \lceil n/\log n \rceil$, for a suitable constant α. Y' is composed by the element y'_m with rank m plus all the other elements equal to y'_m. Y'' is composed by the element y''_m with rank $n-m+1$ plus all the other elements equal to y''_m. To obtain the new sequence we use the in-place, linear time selection and partitioning algorithms in [6,7]. If X is empty, the task left is to sort Z' and Z'', which can be accomplished with any optimal, in-place mergesort (e.g. [10]. Let us denote $Z'Y'$ with M' and $Y''Z''$ with M''. The m pairs of distinct elements $(M'[1], M''[1]), (M'[2], M''[2]), \ldots, (M'[m], M''[m])$ will be used to encode information.

Since the choice of the constant α does not affect the asymptotic complexity of the algorithm, we have reduced our problem to a problem in which we are allowed to use a special *bit memory* with $O(n/\log n)$ bits where each bit can be accessed and modified in constant time but without word-level parallelism.

3.2 Simulating Auxiliary Memory for Permuting

With the internal buffering technique [8], some of the elements are used as placeholders in order to simulate a working area and permute the other elements at lower cost. In our unstable sorting algorithm we use the basic idea of internal buffering in the following way. Using the selection and partitioning algorithms in [6,7], we pass from the original sequence S to ABC with two properties. (i) For any $a \in A$, $b \in B$ and $c \in C$, we have that $a < b < c$. (ii) B is composed of the element b' with rank $\lceil n/2 \rceil$ plus all the other elements equal to b'. We can use BC as an auxiliary memory in the following way. The element in the first position of BC is the *separator element* and will not be moved. The elements in the other positions of BC are placeholders and will be exchanged with (instead of being overwritten by) elements from A in any way the computation on A (in our case the sorting of A) may require. The "emptiness" of any location i of the simulated working area in BC can be tested in $O(1)$ time by comparing the separator element $BC[1]$ with $BC[i]$: if $BC[1] \leq BC[i]$ the ith location is "empty" (that is, it contains a placeholder), otherwise it contains one of the elements in A.

Let us suppose we can sort the elements in A in $O(|A|)$ time using BC as working area. After A is sorted we use the partitioning algorithm in [6] to separate the elements equal to the separator element ($BC[1]$) from the elements greater than it (the computation on A may have altered the original order in BC). Then we just re-apply the same process to C, that is we divide it into $A'B'C'$, we sort A' using $B'C'$ as working area and so forth. Clearly, this process requires $O(n)$ time and when it terminates the elements are sorted. Obviously, we can divide A into $p = O(1)$ equally sized subsequences $A_1, A_2 \ldots A_p$, then sort each one of them using BC as working area and finally fuse them using the in-place, linear time merging algorithm in [10]. Since the choice of the constant p does not affect the asymptotic complexity of the whole process, we have reduced our problem to a new problem, in which we are allowed to use a special *exchange* memory of $O(n)$ locations, where each location can contain *input elements only* (no integers or any other kind of data). Any element can be moved to and from any location of the exchange memory in $O(1)$ time.

3.3 The Reduced Problem

By blending together the basic techniques seen above, we can focus on a reduced problem in which assumption (iv) is replaced by:

(iv) *Only $O(1)$ words of normal auxiliary memory and two kinds of special auxiliary memory are allowed:*
- *(a) A random access* bit memory $\mathscr{B}$ *with $O(n/\log n)$ bits, where each bit can be accessed in $O(1)$ time (no word-level parallelism).*
- *(b) A random access* exchange memory $\mathscr{E}$ *with $O(n)$ locations, where each location can contain* only elements from S *and they can be moved to and from any location of $\mathscr{E}$ in $O(1)$ time.*

If we can solve the reduced problem in $O(n)$ time we can also solve the original problem with the same asymptotic complexity. However, the resulting algorithm will be unstable because of the use of the internal buffering technique with a large pool of placeholder elements.

3.4 The Naive Approach

Despite the two special auxiliary memories, solving the reduced problem is not easy. Let us consider the following naive approach. We proceed as in the normal bucket sorting: one bucket for each one of the n^ϵ range values. Each bucket is a linked list: the input elements of each bucket are maintained in $\mathscr{E}$ while its auxiliary data (e.g. the pointers of the list) are maintained in $\mathscr{B}$. In order to amortize the cost of updating the auxiliary data (each pointer requires a word of $\Theta(\log n)$ bits and $\mathscr{B}$ does not have word-level parallelism), each bucket is a linked list of *slabs* of $\Theta\left(\log^2 n\right)$ elements each ($\mathscr{B}$ has only $O(n/\log n)$ bits). At any time each bucket has a partially full *head slab* which is where any new element of the bucket is stored. Hence, for each bucket we need to store in $\mathscr{B}$ a word of $O(\log\log n)$ bits with the position in the head slab of the last element added. The algorithm proceeds as usual: each element in S is sent to its bucket in $O(1)$ time and is inserted in the bucket's head slab. With no word-level parallelism in $\mathscr{B}$ the insertion in the head slab requires $O(\log\log n)$ time. Therefore, we have an $O(n\log\log n)$ time solution for the reduced problem and, consequently, an unstable $O(n\log\log n)$ time solution for the original problem.

This simple strategy can be improved by dividing the head slab of a bucket into *second level slabs* of $\Theta(\log\log n)$ elements each. As for the first level slabs, there is a partially full, second level head slab. For any bucket we maintain two words in $\mathscr{B}$: the first one has $O(\log\log\log n)$ bits and stores the position of the last element inserted in the second level head slab; the second one has $O(\log\log n)$ bits and stores the position of the last full slab of second level contained in the first level head slab. Clearly, this gives us an $O(n\log\log\log n)$ time solution for the reduced problem and the corresponding unstable solution for the original problem. By generalizing this approach to the extreme, we end up with $O(\log^* n)$ levels of slabs, an $O(n\log^* n)$ time solution for the reduced problem and the related unstable solution for the original problem.

3.5 The Pseudo Pointers

Unlike bit stealing and internal buffering which were known earlier, the pseudo pointers technique has been specifically designed for improving the space complexity in integer sorting problems. Basically, in this technique a set of elements with distinct keys is used as a pool of pre-set, read-only pointers in order to simulate efficiently traversable and updatable linked lists. Let us show how to use this basic idea in a particular procedure that will be at the core of our optimal solution for the reduced problem.

Let d be the number of distinct keys in S. We are given two sets of d input elements with distinct keys: the sets $\mathcal{G}$ and $\mathcal{P}$ of *guides* and *pseudo pointers*, respectively. The guides are given us *in sorted order* while the pseudo pointers form a sequence in arbitrary order. Finally, we are given a multiset $\mathcal{I}$ of d input elements (i.e. two elements of $\mathcal{I}$ can have equal keys). The procedure uses

the guides, the pseudo pointers and the exchange memory to sort the d input elements of $\mathcal{I}$ in $O(d)$ time.

We use three groups of contiguous locations in the exchange memory $\mathscr{E}$. The first group H has n^ϵ locations (one for each possible value of the keys). The second group L has n^ϵ *slots* of two adjacent locations each. The last group R has d locations, the elements of $\mathcal{I}$ will end up here in sorted order. H, L and R are initially empty. We have two main steps.

First. For each $s \in \mathcal{I}$, we proceed as follows. Let p be the leftmost pseudo pointer still in $\mathcal{P}$. If the sth location of H is empty, we move p from $\mathcal{P}$ to $H[s]$ and then we move s from $\mathcal{I}$ to the first location of $L[p]$ (i.e. the first location of the pth slot of L) leaving the second location of $L[p]$ empty. Otherwise, if $H[s]$ contains an element p' (a pseudo pointer) we move s from $\mathcal{I}$ to the first location of $L[p]$, then we move p' from $H[s]$ to the second location of $L[p]$ and finally we move p from $\mathcal{P}$ to $H[s]$.

Second. We scan the guides in $\mathcal{G}$ from the smallest to the largest one. For a guide $g \in \mathcal{G}$ we proceed as follows. If the gth location of H is empty then there does not exist any element equal to g among the ones to be sorted (and initially in $\mathcal{I}$) and hence we move to the next guide. Otherwise, if $H[G]$ contains a pseudo pointer p, there is at least one element equal to g among the ones to be sorted and this element is currently stored in the first location of the pth slot of L. Hence, we move that element from the first location of $L[p]$ to the leftmost empty location of R. After that, if the second location of $L[p]$ contains a pseudo pointer p', there is another element equal to g and we proceed in the same fashion. Otherwise, if the second location of $L[p]$ is empty then there are no more elements equal to g among the ones to be sorted and therefore we can focus on the next guide element.

Basically, the procedure is bucket sorting where the auxiliary data of the list associated to each bucket (i.e. the links among elements in the list) *is implemented by pseudo pointers in* $\mathcal{P}$ instead of storing it explicitly in the bit memory (which lacks of word-level parallelism and is inefficient in access). It is worth noting that the buckets' lists implemented with pseudo pointers are spread over an area that is larger than the one we would obtain with explicit pointers (that is because each pseudo pointer has a key of $\log n^\epsilon$ bits while an explicit pointer would have only $\log d$ bits).

3.6 The Optimal Solution

We can now describe the algorithm, which has three main steps.

First. Let us assume that for any element $s \in S$ there is at least another element with the same key. (Otherwise, we can easily reduce to this case in linear time: we isolate the $O(n^\epsilon)$ elements that do not respect the property, we sort them with the in-place mergesort in [10] and finally we merge them after the other $O(n)$ elements are sorted.) With this assumption, we extract from S two sets $\mathcal{G}$ and $\mathcal{P}$ of d input elements with distinct keys (this can be easily achieved in $O(n)$ time using only the exchange memory $\mathscr{E}$). Finally we sort $\mathcal{G}$ with the optimal in-place mergesort in [10].

Second. Let S' be the sequence with the $(O(n))$ input elements left after the first step. Using the procedure in § 3.5 (clearly, the elements in the sets $\mathcal{G}$ and $\mathcal{P}$ computed in the first step will be the guides and pseudo pointers used in the procedure), we sort each block B_i of S' with d contiguous elements. After that, let us focus on the first $t = \Theta(\log\log n)$ consecutive blocks $B_1, B_2, \ldots, B_t$. We distribute the elements of these blocks into $\leq t$ *groups* $G_1, G_2 \ldots$ in the following way. Each group G_j can contain between d and $2d$ elements and is allocated in the exchange memory $\mathscr{E}$. The largest element in a group is its *pivot*. The number of elements in a group is stored in a word of $\Theta(\log d)$ bits allocated in the bit memory $\mathscr{B}$. Initially there is only one group and is empty. In the ith step of the distribution we scan the elements of the ith block B_i. As long as the elements of B_i are less than or equal to the pivot of the first group we move them into it. If, during the process, the group becomes full, we select its median element and partition the group into two new groups (using the selection and partitioning algorithms in [6,7]). When, during the scan, the elements of B_i become greater than the pivot of the first group, we move to the second group and continue in the same fashion. It is important to notice that the number of elements in a group (stored in a word of $\Theta(\log d)$ bits in the bit memory $\mathscr{B}$) is updated by increments by 1 (and hence the total cost of updating the number of elements in any group is linear in the final number of elements in that group, see [2]). Finally, when all the elements of the first $t = \Theta(\log\log n)$ consecutive blocks $B_1, B_2, \ldots, B_t$ have been distributed into groups, we sort each group using the procedure in § 3.5 (when a group has more than d elements, we sort them in two batches and then merge them with the in-place, linear time merging in [10]). The whole process is repeated for the second $t = \Theta(\log\log n)$ consecutive blocks, and so forth.

Third. After the second step, the sequence S' (which contains all the elements of S with the exclusion of the guides and pseudo pointers, see the first step) is composed by contiguous subsequences $S'_1, S'_2, \ldots$ which are *sorted* and contain $\Theta(d \log\log n)$ elements each (where d is the number of distinct elements in S). Hence, if we see S' as composed by contiguous runs of elements with the same key, we can conclude that the number of runs of S' is $O(n/\log\log n)$. Therefore S' can be sorted in $O(n)$ time using the naive approach described in § 3.4 with only the following simple modification. As long as we are inserting the elements of a single run in a bucket, we maintain the position of the last element inserted in the head slab of the bucket in a word of auxiliary memory (we can use $O(1)$ of them) instead of accessing the inefficient bit memory $\mathscr{B}$ at any single insertion. When the current run is finally exhausted, we copy the position in the bit memory. Finally, we sort $\mathcal{P}$ and we merge $\mathcal{P}$, $\mathcal{A}$ and S' (once again, using the sorting and merging algorithms in [10]).

3.7 Discussion: Stability and Read-Only Keys

Let us focus on the reasons why the algorithm of this section is not stable. The major cause of instability is the use of the basic internal buffering technique in conjunction with large $(\omega(\operatorname{polylog}(n)))$ pools of placeholder elements. This is clearly visible even in the first iteration of the process in § 3.2: after being used to permute A into sorted order, the placeholder elements in BC are left permuted in a completely arbitrary way and their initial order is lost.

4 Reducing Space in any RAM Sorting Algorithm

In this section, we consider the case of sorting integers of $w = \omega(\log n)$ bits. We show a black box transformation from any sorting algorithm on the RAM to a stable sorting algorithm with the same time bounds which only uses $O(1)$ words of additional space. Our reduction needs to modify keys. Furthermore, it requires randomization for large values of w.

We first remark that an algorithm that runs in time $t(n)$ can only use $O(t(n))$ words of space in most realistic models of computation. In models where the algorithm is allowed to write $t(n)$ arbitrary words in a larger memory space, the space can also be reduced to $O(t(n))$ by introducing randomization, and storing the memory cells in a hash table.

Small word size. We first deal with the case $w = \text{polylog}(n)$. The algorithm has the following structure:

1. sort $S[1 \dots n/\log n]$ using in-place stable merge sort [10]. Compress these elements by Lemma 1 gaining $\Omega(n)$ bits of space.
2. since $t(n) = O(n \log n)$, the RAM sorting algorithm uses at most $O(t(n) \cdot w) = O(n \,\text{polylog}(n))$ bits of space. Then we can break the array into chunks of $n/\log^c n$ elements, and sort each one using the available space.
3. merge the $\log^c n$ sorted subarrays.
4. uncompress $S[1 \dots n/\log n]$ and merge with the rest of the array by stable in-place merging [10].

Steps 1 and 4 take linear time. Step 2 requires $\log^c n \cdot t(n/\log^c n) = O(t(n))$ because $t(n)$ is convex and bounded in $[n, n \log n]$. We note that step 2 can always be made stable, since we can afford a label of $O(\log n)$ bits per value.

It remains to show that step 3 can be implemented in $O(n)$ time. In fact, this is a combination of the merging technique from Lemma 2 with an atomic heap [4]. The atomic heap can maintain a priority queue over $\text{polylog}(n)$ elements with constant time per insert and extract-min. Thus, we can merge $\log^c n$ lists with constant time per element. The atomic heap can be made stable by adding a label of $c \log\log n$ bits for each element in the heap, which we have space for. The merging of Lemma 2 requires that we keep track of $O(k/\alpha)$ subarrays, where $k = \log^c n$ was the number of lists and $\alpha = 1/\text{polylog}(n)$ is fraction of additional space we have available. Fortunately, this is only $\text{polylog}(n)$ values to record, which we can afford.

Large word size. For word size $w \geq \log^{1+\varepsilon} n$, the randomized algorithm of [1] can sort in $O(n)$ time. Since this is the best bound one can hope for, it suffices to make this particular algorithm in-place, rather than give a black-box transformation. We use the same algorithm from above. The only challenge is to make step 2 work: sort n keys with $O(n \,\text{polylog}(n))$ space, even if the keys have $w > \text{polylog}(n)$ bits.

We may assume $w \geq \log^3 n$, which simplifies the algorithm of [1] to two stages. In the first stage, a signature of $O(\log^2 n)$ bits is generated for each input value (through hashing), and these signatures are sorted in linear time. Since we are

working with $O(\log^2 n)$-bit keys regardless of the original w, this part needs $O(n \operatorname{polylog}(n))$ bits of space, and it can be handled as above.

From the sorted signatures, an additional pass extracts a subkey of $w/\log n$ bits from each input value. Then, these subkeys are sorted in linear time. Finally, the order of the original keys is determined from the sorted subkeys and the sorted signatures.

To reduce the space in this stage, we first note that the algorithm for extracting subkeys does not require additional space. We can then isolate the subkey from the rest of the key, using shifts, and group subkeys and the remainder of each key in separate arrays, taking linear time. This way, by extracting the subkeys instead of copying them we require no extra space. We now note that the algorithm in [1] for sorting the subkeys also does not require additional space. At the end, we recompose the keys by applying the inverse permutation to the subkeys, and shifting them back into the keys.

Finally, sorting the original keys only requires knowledge of the signatures and *order* information about the subkeys. Thus, it requires $O(n \operatorname{polylog}(n))$ bits of space, which we have. At the end, we find the sorted order of the original keys and we can implement the permutation in linear time.

Acknowledgements. Our sincere thanks to Michael Riley and Mehryar Mohri of Google, NY who, motivated by manipulating transitions of large finite state machines, asked if bucket sorting can be done in-place in linear time.

References

1. Andersson, A., Hagerup, T., Nilsson, S., Raman, R.: Sorting in linear time? Journal of Computer and System Sciences 57(1), 74–93 (1998)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
3. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. System Sci. 47, 424–436 (1993)
4. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. System Sci. 48(3), 533–551 (1994)
5. Han, Y., Thorup, M.: Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In: FOCS, pp. 135–144. IEEE Computer Society, Los Alamitos (2002)
6. Katajainen, J., Pasanen, T.: Stable minimum space partitioning in linear time. BIT 32(4), 580–585 (1992)
7. Katajainen, J., Pasanen, T.: Sorting multisets stably in minimum space. Acta Informatica 31(4), 301–313 (1994)
8. Kronrod, M.A.: Optimal ordering algorithm without operational field. Soviet Math. Dokl. 10, 744–746 (1969)
9. Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in O($\log^2$ n) time. Journal of Computer and System Sciences 33(1), 66–74 (1986)
10. Salowe, J., Steiger, W.: Simplified stable merging tasks. Journal of Algorithms 8(4), 557–571 (1987)

Fast Low Degree Connectivity of Ad-Hoc Networks Via Percolation

Emilio De Santis[1], Fabrizio Grandoni[2], and Alessandro Panconesi[2]

[1] Dipartimento di Matematica, Sapienza Università di Roma,
P.le Aldo Moro 2, 00185 Roma, Italy
desantis@mat.uniroma1.it
[2] Dipartimento di Informatica, Sapienza Università di Roma,
Via Salaria 113, 00198 Roma, Italy
{grandoni,ale}@di.uniroma1.it

Abstract. Consider the following classical problem in ad-hoc networks: n devices are distributed uniformly at random in a given region. Each device is allowed to choose its own transmission radius, and two devices can communicate if and only if they are within the transmission radius of each other. The aim is to (quickly) establish a connected network of low average and maximum degree.

In this paper we present the first efficient distributed protocols that, in poly-logarithmically many rounds and with high probability, set up a connected network with $O(1)$ average degree and $O(\log n)$ maximum degree. This is asymptotically the best possible.

Our algorithms are based on the following result, which is a non-trivial consequence of classical percolation theory: suppose that all devices set up their transmission radius in order to reach the K closest devices. There exists a universal constant K (independent of n) such that, with high probability, there will be a unique giant component, i.e. a connected component of size $\Theta(n)$. Furthermore, all remaining components will be of size $O(\log^2 n)$. This leads to an efficient distributed probabilistic test for membership in the giant component, which can be used in a second phase to achieve full connectivity.

Preliminary experiments suggest that our approach might very well lead to efficient protocols in real wireless applications.

1 Introduction

In this paper we study a geometric random graph model that has interesting applications to wireless networking. We are given n points distributed uniformly at random within the unit square. Each point v is connected via a directed arc to the *closest* $k(v)$ points, according to the Euclidean distance, where $k(v)$ is a positive integer value. Given this directed graph we define an undirected graph G with the same vertex set as follows: $vw \in E(G)$ if and only if there is a directed arc from v to w and viceversa. Henceforth, we will refer to the the points also as *nodes* or *devices*.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 206–217, 2007.

The question that we study in this paper is how to determine the value of the $k(v)$'s in order to meet two conflicting goals: G should be connected, but its average degree should be as small as possible. Moreover, the maximum degree should also be small. In this paper we give two efficient (i.e. poly-logarithmic) distributed algorithms that set up a connected network G in such a way that (a) the expected degree of a node is constant and (b) the maximum degree is $O(\log n)$ (resp. $O(\log^2 n)$) with high probability. The number of communication rounds needed is $O(\log^3 n)$ (resp. $O(\log^2 n)$).

These results appear to be relevant to wireless networking. Our model of connectivity for G is the most realistic from the point of view of wireless applications since the communication primitives of standards such as IEEE 802.11 and Bluetooth rely on ack messages, and therefore a communication link really exists only when both nodes are within transmission radius of each other. Our algorithms are very simple and we give experimental evidence that they might very well admit efficient wireless implementations. Limiting the degree of nodes can be beneficial in many ways. For instance, in security applications, nodes exchange keys and run cryptographic protocols with their neighbors (see, for for instance, [13]). Limiting the degree reduces the amount of traffic and computation. Moreover, the transmission radius of v is set in order to reach its $k(v)$ closest neighbors. Hence, the larger $k(v)$, the larger the power v needs. Limiting the $k(v)$'s thus reduces the overall transmission power and translates in longer network lifetimes. In particular, we can show that while (a) the optimal power consumption is, with high probability, proportional to the area of the region within which the nodes are randomly distributed, (b) the expected power consumption to sustain the network with our approach is order of the area and hence it is in some sense optimal. (This is a consequence of our main results and its proof is omitted from this extended abstract). Probably the most important benefit is that, by bounding $k(v)$ and by setting the transmission power accordingly, interference is kept under control: The lower a node's degree, the lower the number of neighbors affected by a transmission and, consequently, the lower the number of possible packet collisions and corresponding retransmission (see, for instance, [1]). Note that our high probability bound on the maximum degree ensures that not only things are good on average, but also that no node will be penalized too much.

Let us now describe our algorithms. Probably the simplest distributed algorithm one can think of is the following: set beforehand $k(v) = K$, for all nodes v and for a suitable constant K (see [5,6,10,11] for experimental results). Unfortunately, there is no constant K which guarantees connectivity with probability going to 1 as n grows. To reach that goal, K must grow like $\log n$ [12].

If points can communicate, the situation changes. Indeed, Kucera [7] gives a protocol to decide $k(v)$, for all v, that sets up a connected network of expected constant degree and maximum degree $O(\log n)$. The result however is existential in flavor: the protocol requires nodes to explore linear-size components of the network, linearly many times, making it completely impractical. Our faster protocols are based on the following insight.

Theorem 1. *There is a universal constant K, independent of n, such that, if all the devices set $k(v) = K$, with probability going to 1 as n grows, the network has the following special structure: (a) there is a unique giant component containing $\Theta(n)$ nodes; (b) all other components have size $O(\log^2 n)$.*

This theorem says that it is possible to set up a giant component in a very simple way, a useful fact by itself (e.g. for coverage applications). It also says that there is an efficient distributed test for membership to the giant component: a node belongs to the unique giant component if and only if it belongs to a component with more than (order of) $\log^2 n$ nodes.

Given this, the following strategy is very natural. Devices that discover to be trapped inside small components increase their transmitting power in order to reach a device that belongs to the giant component. A node in the giant component that is contacted in this way will respond, setting its power in order to reach the calling node. We shall refer to this as Algorithm A.

Theorem 2. *Algorithm A sets up a network in which the expected number of neighbors of each device is constant. Furthermore, with probability going to 1 as n grows, the network is connected and its maximum degree is $O(\log^2 n)$. The number of communication rounds required is $O(\log^2 n)$.*

This result gives an exponential speed up with respect to [7]. We can improve the bound on the maximum degree at the expense of an increased communication cost. Suppose that each device v belonging to a small component increases its transmitting power a bit at a time, each time checking if it has reached a node in the giant component. Nodes closer to the giant component will join it first. Nodes farther away might be able to connect to such closer nodes, rather than expanding their radius all the way to the closest node in the original giant component. In the next section we will give a precise description of this, referred to as Algorithm B.

Theorem 3. *Algorithm B sets up a network in such a way that the expected number of neighbors of each device is constant. Furthermore, with probability going to 1 as n grows, the network is connected and its maximum degree is $O(\log n)$. The number of communication rounds required is $O(\log^3 n)$.*

Preliminary experiments show that our approach might very well lead to efficient protocols in real wireless applications.

2 The Algorithms

The input to the algorithms consists of n devices that are spread uniformly at random within the unit box. The value of n is known to the devices. We assume that the network is synchronous and that in one communication round each device is able to send messages to all neighbors and to receive messages from all of them. The running time of the protocols is given by the number of such communication rounds. Each device is initially marked as *lacustrine*. Algorithm A has two constant parameters K and φ, and works as follows.

Phase 1: Every device v sets its own transmission radius in order to reach the closest $k(v) := K$ neighbors (all the devices if $n < K$).
Phase 2: Every device v explores its own connected component, denoted as $C(v)$. If $|C(v)| > C = \varphi \log^2 n$, v marks itself as *continental*. Every lacustrine device v increases $k(v)$ in order to reach the closest continental device, denoted as $s(v)$. Device $s(v)$ responds by increasing its transmission radius in order to reach v (if this is not already the case).

Algorithm B, has a third constant parameter $\mu > 0$, and works as follows:

Phase 1: As in Algorithm A.
Phase 2: Repeat $\mu \log n$ many times: Let v be lacustrine. If $|C(v)| > C = \varphi \log^2 n$ then v marks itself as *continental*. Otherwise, v increases $k(v)$ by one, in order to reach the next closest device $s(v)$. If $s(v)$ is continental, it responds by increasing its transmission radius in order to reach v.

The mapping $s(v)$ is, for all practical purposes, well-defined since almost surely all pairwise distances are different. The constants K and φ (independent of n) ensure that, with high probability, at the end of the first phase there is a unique giant component of $\Theta(n)$ points, while all the other components contain $\varphi \log^2 n$ points. Observe that Phase 1 does not require any global information, such as the value of n. We shall refer to a round of Phase 2 of Algorithm B as an *expansion* round. The constant μ ensures that, with high probability, within $\mu \log n$ expansion rounds all the nodes become continental. As a consequence, Algorithms A and B achieve connectivity with high probability. Moreover they require $O(\log^2 n)$ and $O(\log^3 n)$ communication rounds, respectively.

3 Overview

Since the proof of Theorems 1, 2, and 3 is rather involved we first give an overview. The basic idea is to reduce our connectivity problem to site percolation in a finite box (for an introduction to percolation, see, e.g., [3,9]). It is known that in the supercritical phase, with high probability there is a unique giant cluster in the box and that its complement consists of small regions each containing $O(\log^2 n)$ sites (see, among others, [2,3,4]). In the following we shall refer to the maximal regions in the complement of the giant cluster as *lakes*. The reduction will ensure that the unique giant cluster in the box will correspond to a unique giant component of points, and that the remaining components of points are trapped inside lakes, each containing $O(\log^2 n)$ points. This is the situation at the end of Phase 1 (with high probability).

The reduction to site percolation is achieved via several intermediate steps. The first is to replace the uniform distribution of points with a Poisson distribution, to exploit the strong independence properties of the latter. In particular, unlike the uniform distribution, the Poisson distribution ensures that the configuration of points in one region does not affect the distribution of points of any other disjoint region. There are some standard and rather general ways to

connect the two settings, but here we will make use of a coupling construction that gives stronger bounds than these general tools. The configurations of points given by the mentioned Poisson processes is referred to as scenario $\mathcal{A}$.

We introduce next a first percolation problem, scenario $\mathcal{B}$, by subdividing the unit square into a grid of non-overlapping square cells. The area of each cell is such that the expected number of points inside it is a constant parameter α. This parameter is crucial for the whole construction. A cell is *good* if the number of points that it contains is in $[\frac{\alpha}{2}, 2\alpha]$.

Scenario $\mathcal{B}$ is a Bernoulli field but unfortunately clusters of good cells do not translate necessarily into connected components of points. Therefore another percolation problem, scenario $\mathcal{C}$, is introduced by defining a cell i *open* if it is good and moreover all cells within distance D from i are good. The value of D is a constant, independent of n. The definition is such that the points belonging to a cluster of open cells form themselves a component of points.

The problem with scenario $\mathcal{C}$ is that it is not a Bernoulli field– knowing that a cell i is open or closed alters the distribution of neighboring cells. However the mentioned dependence only involves cells at distance at most $h = 2D$ from i, that is the field is *h-dependent* (see [3]). Therefore a new scenario $\mathcal{D}$ is introduced. Scenario $\mathcal{D}$ is given by a general construction of [8]. This construction translates scenario $\mathcal{C}$ into a Bernoulli field (i.e. $\mathcal{D}$) that is stochastically dominated by $\mathcal{C}$: if a cell is open in scenario $\mathcal{D}$ then it is also open in scenario $\mathcal{C}$. It follows that if a giant cluster of open cells exists with probability p in scenario $\mathcal{D}$, the same cluster exists in scenario $\mathcal{C}$ with probability at least p. Essentially, scenario $\mathcal{C}$ ensures that the unique giant component of cells that, with high probability, exists in it translates into a connected component of points in scenario $\mathcal{A}$, and that all other components are small. While scenario $\mathcal{D}$ is used to compute the probability that these events take place.

The probability that sites are on or off in the various scenarios depends on the value of the constant K of the protocol. We will fix K in such a way that a unique giant cluster of open cells exists in scenario $\mathcal{D}$ with high probability. By construction, this translates into a giant component of points in scenario $\mathcal{A}$, henceforth denoted as $\mathcal{G}$.

To ensure that $\mathcal{G}$ is unique in scenario $\mathcal{A}$ we make use of the definition of open cells of scenario $\mathcal{C}$ which ensures that points trapped inside lakes cannot connect to points in other lakes, bypassing $\mathcal{G}$.

Remark 1. By setting the radius of each point to $\sim n^{-1/2}$ we would obtain a simpler reduction to site percolation to show the emergence of a giant component. Our reduction however is independent of n, showing that a giant component can be created with no global information at all. This might be of independent interest.

3.1 Preliminaries

As mentioned, in scenarios $\mathcal{B}, \mathcal{C}$, and $\mathcal{D}$, we consider a partition of the unit square into a grid of non-overlapping square cells of the same size. The number of cells is

$m = k^2$, where $k := \lfloor \sqrt{\frac{n}{\alpha}} \rfloor$, and α is a constant. This partition naturally induces a mesh, where the nodes are the cells and each cell has (at most) four neighbors: the cells on the left, right, top and bottom. Let $i_{x,y}$ be the cell in position (x, y) in the grid. The *distance* between i_{x_1,y_1} and i_{x_2,y_2} is $\max\{|x_1 - x_2|, |y_1 - y_2|\}$. The *star-neighbors* of cell i are the cells at distance one from i. We call *cluster* a connected component of cells, and *star-cluster* a connected component of cells with respect to star-neighborhood. We will use this distance in the mesh, while we will use the Euclidean distance when talking about points in the unit square.

A *giant* cluster is a cluster of open cells which contains at least $\delta\, m$ cells, for a given constant $\delta \in (0, 1]$. Assuming a unique giant cluster (an event that we will show happening with high probability), a *lake* is a maximal star-cluster in the complement of the giant cluster. A *giant component* is a connected component of points of linear (in n) size in the network set-up by the protocol. With $|X|$ we denote either the number of cells of X or the number of points of X, depending on whether X is a cluster or a component, respectively.

4 Emergence of a Giant Component

In this section we show that after Phase 1 of the algorithm $|\mathcal{G}| = \Theta(n)$ with high probability.

As outlined previously we consider four different scenarios. In scenario $\mathcal{A}$ points are placed in the unit box by means of a Poisson process. More precisely, we consider two Poisson processes P_0 and P_t. Process P_0 has parameter $\mu_0 := n - \epsilon\, n$, where ϵ is a small positive constant, say $\epsilon = \frac{1}{4}$. Process P_t is built on top of P_0 by adding to it a new independent Poisson process ΔP with parameter $2\,\epsilon\, n$. It is well-known that P_t is a Poisson process with parameter $\mu_t := \mu_0 + 2\,\epsilon\, n = n + \epsilon\, n$. We then define a sequence of point processes $\{Q_i\}$ sandwiched between P_0 and P_t. Starting from $Q_0 := P_0$, Q_{i+1} is given by Q_i by adding one point chosen uniformly at random in $P_t - Q_i$.

Our reduction to site percolation will apply simultaneously to all Q_i's, showing the existence of a unique giant component in scenario $\mathcal{A}$ for each Q_i with high probability. Each Q_i generates points uniformly in the box (conditioned on the given number of points). The next lemma shows that, with high probability, one of the Q_i will generate exactly n points. As a consequence, if something holds for all Q_i's of scenario $\mathcal{A}$ simultaneously, it also holds for the original n-points problem.

Lemma 1. *Let N_0 and N_t be the Poisson variables relative to P_0 and P_t, respectively. There is a positive constant γ (independent of n) such that*

$$\Pr\left(\overline{\{N_0 \leq n \leq N_t\}}\right) \leq e^{-\gamma n}.$$

Proof. (Sketch) Apply the large deviation principle to $\{N_0 > n\}$ and to $\{n < N_t\}$.

We now define scenario $\mathcal{B}$. Let us subdivide the unit square into a grid of $m = k^2$ non-overlapping square cells, where $k := \lfloor \sqrt{\frac{n}{\alpha}} \rfloor$, and α is a constant. Note that

$m = \Theta(n)$ and the expected number of points in a cell is (roughly) α. The parameter α plays a crucial role in the whole proof. This parameter should be thought of as a large constant. Its value will be fixed later.

Definition 1. *A cell is* good *if the number of points in the cell given by both P_0 and P_t is in $[\frac{\alpha}{2}, 2\alpha]$. The cell is* bad *otherwise.*

In scenario $\mathcal{B}$ we define a site percolation problem with a Bernoulli field, where the good cells will be the on sites in the finite box. Note that if a cell is good then its number of points is in $[\frac{\alpha}{2}, 2\alpha]$ for all Q_i's of scenario $\mathcal{A}$. By construction, cells are good independently of each other and the probability of being good is the same for all cells.

Lemma 2. *Let p_α be the probability that a cell is good. Then* $\lim_{\alpha\to\infty} p_\alpha = 1$.

Proof. Apply the standard large deviation principle to the Poisson random variables corresponding to the number of points given by P_0 and P_t in the cell considered.

We would like to show that large connected clusters of good cells in scenario $\mathcal{B}$ give raise to large connected components of points in scenario $\mathcal{A}$. This however is not true. This motivates the next scenario $\mathcal{C}$. In it we consider another site percolation problem which is not, however, independent. Let $D \geq 3$ be a constant to be fixed later.

Definition 2. *A cell i is* open *if i and all the cells at distance at most D from i are good. The cell is* closed *otherwise.*

We now choose K and D in order to enforce the following two properties: First, if we have a giant cluster of open cells in scenario $\mathcal{C}$, then the points inside these cells belong to a giant component $\mathcal{G}$ of scenario $\mathcal{A}$ for each Q_i. Second, components other than $\mathcal{G}$ will be trapped inside lakes (delimited by closed cells), i.e. points inside distinct lakes cannot establish links among them directly, bypassing $\mathcal{G}$. The first property is guaranteed by choosing

$$K := 7^2(2\alpha) = 98\,\alpha$$

(the maximum number of points in the cells at distance at most 3 from a given open cell). This choice of K ensures that the transmission radius of every point in an open cell will reach all points in neighboring cells. Thus, if two neighboring cells are open the points in them will form a clique. Observe that, for similar reasons, any point inside a cell at distance at most $D-3$ from an open cell i belongs to the same connected component to which the points in i belong. This implies that, by choosing D such that $2(D-3)\alpha/2 > K$, say $D = 102$, also the second property is ensured (see the proof of Lemma 5). We have not tried to optimize the values of D and K.

In scenario $\mathcal{C}$ we have the desired translation of connectivity– if we have a cluster of open cells then all points belonging to these cells form a connected component of points in scenario $\mathcal{A}$ (for all Q_i's). Moreover, the probability q_α

that a cell is open satisfies $q_\alpha \geq p_\alpha^{(2D+1)^2}$ and thus $\lim_{\alpha\to\infty} q_\alpha = 1$. Unfortunately scenario $\mathcal{C}$ is not a Bernoulli field. Definition 2 however ensures that it is h-dependent with $h = 2D$ (the probability that a cell is open is independent from what happens in cells at distance $2D+1$ or larger).

Therefore we introduce a fourth scenario $\mathcal{D}$ that is a Bernoulli field. The connection between scenarios $\mathcal{C}$ and $\mathcal{D}$ is given by a very general theorem of [8]. The theorem states that there is a coupling between scenario $\mathcal{C}$ and a Bernoulli field, referred to as scenario $\mathcal{D}$, with site probability r_α such that: (a) If q_α goes to 1 so does r_α, and hence $\lim_{\alpha\to\infty} r_\alpha = 1$; and, (b) If a cell is open in scenario $\mathcal{D}$ the same cell is open in scenario $\mathcal{C}$. Therefore, if we have a giant cluster in scenario $\mathcal{D}$, the same cells form a cluster also in scenario $\mathcal{C}$. In turn, all points inside these cells will be connected in scenario $\mathcal{A}$.

Scenario $\mathcal{D}$ allows us to estimate the probability of relevant events. A result of Deuschel and Pisztora [2] ensures that, for every constant $\delta \in (0,1)$ there is a value of $r_\alpha < 1$ such that, with probability at least $1 - e^{-\gamma\sqrt{m}}$ there is a unique giant cluster of at least δm open cells in scenario $\mathcal{D}$, for some constant $\gamma > 0$. The following theorem and corollary summarize the discussion above.

Theorem 4. *Let $\mathcal{G}$ denote a maximum cardinality component of points at the end of Phase 1 of the algorithm. For every $c \in (0, \frac{1}{2})$ there is a choice of $\alpha > 0$, and so a corresponding choice of K, such that*

$$\Pr(|\mathcal{G}| \leq cn) \leq 2\, e^{-\xi\sqrt{n}}$$

where $\xi > 0$ is a constant independent of n.

Proof. Let $m \simeq \frac{n}{\alpha}$ be the number of cells in scenario $\mathcal{D}$, and let C be a maximum size cluster in scenario $\mathcal{D}$. By [2], for any given $\delta \in (0,1)$ there is a value of α such that $\Pr(|C| \leq (1-\delta)m) \leq e^{-\gamma\sqrt{m}}$. The same set of cells is open in scenario $\mathcal{C}$. By definition, an open cell contains at least $\frac{\alpha}{2}$ points. Thus, C corresponds to a giant component of at least $(1-\delta)m\frac{\alpha}{2}$ points in scenario $\mathcal{A}$ (for all Q_i's). Recalling Lemma 1 and by choosing $\delta < 1 - 2c$, we have

$$\Pr(|\mathcal{G}| \leq cn) \leq \Pr(|C| \leq 2cn/\alpha) + \Pr(\overline{\{N_0 \leq n \leq N_t\}}) \leq 2\, e^{-\xi\sqrt{n}},$$

for some constant $\xi > 0$ and n large enough. This follows from the fact that if the condition $\{N_0 \leq n \leq N_t\}$ does not hold we give up, while we pursue the construction of the 4 scenarios only if it holds.

Remark 2. By choosing ϵ appropriately in the definition of the two Poisson processes P_0 and P_t of scenario $\mathcal{A}$, and by defining a cell to be good if its number of points in both P_0 and P_t is in the interval $[(1-\epsilon')\alpha, (1+\epsilon')\alpha]$, the size of $\mathcal{G}$ can be made arbitrarily close to 1, for a proper choice of ϵ'.

Corollary 1. *For every $c \in (0,1)$ there exist constants $K > 0$ and $\gamma > 0$ such that, if every point v sets $k(v) = K$, then $\Pr[|\mathcal{G}| < cn] \leq e^{-\gamma\sqrt{n}}$.*

Proof. It follows from the proof of Theorem 4 and Remark 2.

5 Uniqueness of the Giant Component

In this section we show that at the end of Phase 1 of the algorithm, with probability $1 - o(1)$, there is a unique giant component $\mathcal{G}$ with linearly many points, and that all other components contain $O(\log^2 n)$ points.

The next lemma bounds the number of cells of a lake in scenario $\mathcal{C}$.

Lemma 3. *For any lake L of scenario $\mathcal{C}$, $\Pr(|L| > k) \leq e^{-\gamma\sqrt{k}}$, where $\gamma > 0$ is a constant and $|L|$ is the number of cells of L.*

Proof. It is well-known that for a Bernoulli field, in the super-critical phase for percolation, if we take any star-cluster S in the complement of the giant cluster then $\Pr(|S| > k) \leq e^{-\gamma\sqrt{k}}$, for some constant $\gamma > 0$ [3]. Therefore the same holds for any lake of scenario $\mathcal{D}$. Now, by the monotonicity implied by the coupling construction of [8] that relates scenario $\mathcal{C}$ and scenario $\mathcal{D}$, the same bound holds for L in scenario $\mathcal{C}$ (lakes can only be smaller).

The next, crucial lemma bounds the number of points of each lake in scenario $\mathcal{A}$. The difficulty is to analyze the dependencies carefully– knowing that a cell is closed/open affects not only the distribution of points inside this cell, but also that of neighboring cells.

Lemma 4. *Let Z_i be the number of points in cell i, and let L be a lake in scenario $\mathcal{C}$ with n points. Then, for large enough number of points n, there is a constant $\gamma > 0$ such that*

$$\Pr(\sum_{i \in L} Z_i > h) \leq e^{-\gamma\sqrt{h}}.$$

Proof. Let $B := (B_1, \ldots, B_m)$ be the random vector denoting which cells are good or bad, and $b = (b_1, \ldots, b_m)$ any particular such configuration. Then

$$\begin{aligned}
\Pr(\sum_{i \in L} Z_i > h) &= \sum_k \Pr\left(\sum_{i \in L} Z_i > h \mid |L| = k\right) \Pr(|L| = k) \\
&= \sum_k \sum_b \Pr\left(\sum_{i \in L} Z_i > h \mid |L| = k, B = b\right) \Pr(B = b \mid |L| = k) \Pr(|L| = k) \\
&= \sum_k \sum_b \Pr(\sum_{i \in L} Z_i > h \mid B = b) \Pr(B = b \mid |L| = k) \Pr(|L| = k).
\end{aligned}$$

The last equality follows, since if we know B we also know the size of L. We now focus on the term $\Pr(\sum_{i \in L} Z_i > h \mid B = b)$. We will show that we can replace the variables $(Z_i \mid B = b)$ with a set of i.i.d. variables that stochastically dominate them and that obey the large deviation principle.

The Poisson process can be realized as the product of m independent Poisson processes, each operating inside a cell. This implies that if we have a set of events E_i where each event depends only on what happens in cell i, then $\Pr(\cap_i E_i) = \prod_i \Pr(E_i)$. Thus, we have

$$\Pr(\cap_i\{Z_i = h_i\}|B = b) = \frac{\Pr(\cap_i\{Z_i = h_i, B_i = b_i\})}{\Pr(\cap_i\{B_i = b_i\})} = \frac{\prod_i \Pr(Z_i = h_i, B_i = b_i)}{\prod_i \Pr(B_i = b_i)}$$
$$= \prod_i \Pr(Z_i = h_i|B_i = b_i).$$

If we define $X_i = (Z_i|B_i = \text{good})$ and $Y_i = (Z_i|B_i = \text{bad})$, it follows that $\sum_i(Z_i|B)$ has the same law of the sum of independent variables each of which is X_i or Y_i depending on whether cell i is good or bad. Let us define a collection of i.i.d. random variables W_i's each of which has the distribution of $(Z_i|Z_i > 2\alpha)$. Each W_i stochastically dominates both X_i and Y_i so that

$$\Pr(\sum_{i\in L} Z_i > h \mid B = b) \le \Pr(\sum_{i\in L} W_i > h),$$

for each configuration b. Moreover the W_i obey the large deviation principle, i.e. the probability of large deviations from the mean is exponentially small. We thus have, for $\beta < 1/E[W_1]$,

$$\begin{aligned}
\Pr(\sum_{i\in L} Z_i > h) &= \sum_k \sum_b \Pr(\sum_{i\in L} Z_i > h \mid B = b) \Pr(B = b \mid |L| = k) \Pr(|L| = k) \\
&\le \sum_k \sum_b \Pr(\sum_{i\le k} W_i > h) \Pr(B = b \mid |L| = k) \Pr(|L| = k) \\
&= \sum_k \Pr(\sum_{i\le k} W_i > h) \Pr(|L| = k) \\
&= \sum_{k\le\beta h} \Pr(\sum_{i\le k} W_i > h) \Pr(|L| = k) + \sum_{k>\beta h} \Pr(\sum_{i\le k} W_i > h) \Pr(|L| = k) \\
&\le \sum_{k\le\beta h} \Pr(\sum_{i\le\beta h} W_i > h) \Pr(|L| = k) + \sum_{k>\beta h} \Pr(\sum_{i\le k} W_i > h) \Pr(|L| = k) \\
&\le \sum_{k\le\beta h} \Pr(\sum_{i\le\beta h} W_i > h) + \sum_{k>\beta h} \Pr(|L| = k) \\
&= \beta h \Pr(\sum_{i\le\beta h} W_i > h) + \sum_{k>\beta h} \Pr(|L| = k) \\
&\le \beta h\, e^{-\gamma_1 h} + \sum_{k>\beta h} e^{-\gamma_2\sqrt{k}} \le e^{-\gamma\sqrt{h}}.
\end{aligned}$$

The next lemma shows that, for any given Q_i of scenario $\mathcal{A}$ components of points inside distinct lakes cannot hook up together, by-passing $\mathcal{G}$.

Lemma 5. *Let u and v be points contained in two distinct lakes of scenario $\mathcal{C}$. Unless they both belong to $\mathcal{G}$ they are disconnected.*

Proof. Let us assume by contradiction that u and v are connected in scenario $\mathcal{A}$ without being connected to $\mathcal{G}$. Since u and v belong to different lakes, they must be separated by a portion of the giant cluster. Let i be an open cell and let v_i be a point inside it. By definition of open, if v_j is a point inside a cell j within distance $D - 3$ from i, then v_i and v_j belong to the same connected component. In particular, if v_i belongs to the giant component, so does v_j. Thus,

u and v must be separated by at least $2(D-3)$ good cells. But each good cell contains at least $\alpha/2$ points, and $2(D-3)\alpha/2 = 102\alpha - 3\alpha > 98\alpha = K$, which is a contradiction.

The following lemma immediately follows from Lemmas 1, 4, and 5, and concludes the proof of Theorem 1.

Lemma 6. *Consider the following event $\mathcal{E}$: at the end of Phase 1 there is a unique giant component containing at least $c\,n$ points while the remaining components are trapped inside lakes, with each lake containing at most $\varphi \log^2 n$ points. For every constant $c \in (0,1)$, and for n sufficiently large, $\Pr[\overline{\mathcal{E}}] \leq 2/n^d$ where $d = \gamma\sqrt{\varphi} - 1$ and $\gamma > 0$ is a constant.*

Proof. By Corollary 1, for every $c \in (0,1)$, there exist constants γ_1 and $\hat{K}$, such that, when the algorithm is run with parameter $K \geq \hat{K}$, the probability that, at the end of phase 1, there is no component with at least $c\,n$ points, is at most $2e^{-\gamma_1\sqrt{n}}$. By Lemma 4 and the union bound, the probability that there exists a lake with more than $\varphi \log^2 n$ points is at most $ne^{-\gamma_2\sqrt{\varphi \log^2 n}} = n^{-\gamma_2\sqrt{\varphi}+1}$. Thus $\Pr[\overline{\mathcal{E}}] \leq 2e^{-\gamma_1\sqrt{n}} + n^{-\gamma_2\sqrt{\varphi}+1} \leq 2/n^d$ for n large enough.

Therefore, by choosing φ large enough, we can bound $\Pr[\overline{\mathcal{E}}]$ with any inverse polynomial.

6 Expected and Maximum Degree

Let us now consider the degree of the nodes at the end of the algorithms. We first analyze the maximum degree.

Lemma 7. *With probability $1-o(1)$ the final maximum degree is $O(\log^2 n)$ with Algorithm A.*

Proof. (Sketch) For a lacustrine node the bound follows from Lemma 6, observing that before reaching the closest continental node, lacustrine nodes that increase their radius will cover at most their own lake, which is of size $O(\log^2 n)$ with high probability. We omit the proof of the bound for continental nodes, which is analogous.

For lack of space we do not give the proof of the following lemma.

Lemma 8. *With probability $1-o(1)$ the final maximum degree is $O(\log n)$ with Algorithm B.*

Lemma 9. *With both Algorithms A and B, the expected final degree of a point is bounded by a constant.*

Proof. (Sketch) We bound the degree for Algorithm A. Basically the same proof holds for Algorithm B as well. Consider first the expected degree of any lacustrine point v. Let L be the lake containing v at the end of the first Phase and let w be

the point of the initial giant component $\mathcal{G}$ closest to v. By Lemma 5 the value of $k(v)$ is bounded by $1 + \sum_{i \in L} Z_i$ since in the worst case, v will capture the points in L plus w. By Lemma 4,

$$E[\sum_{i \in L} Z_i] \leq \sum_h h \Pr(\sum_{i \in L} Z_i \geq h) \leq \sum_h h e^{-\gamma \sqrt{h}} < \infty.$$

The growth of the degree of continental nodes can be bounded in a similar way, and thus we omit the proof in this extended abstract.

This ends the proof of Theorems 2 and 3. We remark that the probability that the protocols fail can be made as small as n^{-d}, for any constant $d > 0$, by a suitable choice of constants that appear in the analysis.

References

1. Blough, D., Leoncini, M., Resta, G., Santi, P.: The k-Neighbors Approach to Interference Bounded and Symmetric Topology Control in Ad Hoc Networks, IEEE Trans. on Mobile Computing
2. Deuschel, J.-D., Pisztora, A.: Surface order large deviations for high-density percolation. Probability Theory and Related Fields 104(4), 467–482 (1996)
3. Grimmet, G.: Percolation. Springer, Heidelberg (1989)
4. Häggström, O., Meester, R.: Nearest neighbor and hard sphere models in continuum percolation. Random Structures and Algorithms 9, 295–315 (1996)
5. Hou, T., Li, V.: Transmission range control in multihop packet radio networks. IEEE Transactions on Communications COM-34, 38–44 (1986)
6. Kleinrock, L., Silvester, J.: Optimum transmission radii for packet radio networks or why six is a magic number. In: IEEE National Telecommunication Conference, pp. 431–435 (1978)
7. Kucera, L.: Low degree connectivity in ad-hoc networks. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 203–214. Springer, Heidelberg (2005)
8. Liggett, T.M., Schonmann, R.H., Stacey, A.M.: Domination by product measures. Annals of Probability 25(1), 71–95 (1997)
9. Meester, R., Roy, R.: Continuum Percolation. Cambridge University Press, Cambridge (1996)
10. Ni, J., Chandler, S.A.G.: Connectivity Properties of a Random Radio Network. IEE Prec. Commun. 141, 289–296 (1994)
11. Takagi, H., Kleinrock, L.: Optimal transmission ranges for randomly distributed packet radio terminals. IEEE Transactions on Communications COM-32(3), 246–257 (1984)
12. Xue, F., Kumar, P.: The number of neighbors needed for connectivity of wireless networks. Wireless Networks 10(2), 169–181 (2004)
13. Zhu, S., Setia, S., Jajodia, S.: LEAP: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks. In: Proc. of the 10th ACM Conference on Computer and Communications Security (CCS '03), Washington, D.C. (October 2003)

Order Statistics in the Farey Sequences in Sublinear Time

Jakub Pawlewicz

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, Poland
pan@mimuw.edu.pl

Abstract. The paper presents the first sublinear algorithm for computing order statistics in the Farey sequences. The algorithm runs in time $O(n^{3/4} \log n)$ and in space $O(\sqrt{n})$ for Farey sequence of order n. This is a significant improvement to the algorithm from [1] that runs in time $O(n \log n)$.

1 Introduction

The Farey sequence of order n (denoted $\mathcal{F}_n$) is the increasing sequence of all irreducible fractions from interval $[0, 1]$ with denominators less than or equal to n. The Farey sequences have numerous interesting properties and they are well known in the number theory and in the combinatorics. They are deeply investigated in [2]. In this paper we study the following algorithmic problem. For given positive integers n and k compute the k-th element of the Farey sequence of order n. This problem is known as *order statistics* problem.

The solution to the order statistics problem is based on a solution to a related *rank problem*, i.e. the problem of finding the rank of a given fraction in the Farey sequence. Both, the order statistics problem and the rank problem, can be easily solved in quadratic time by listing all elements of the sequence (see [2, Problem 4-61] and Section 2.1).

Faster solutions for order statistics are possible by reducing the main problem to the rank problem. The roughly linear time algorithm is presented in [1]. The authors present a solution of the rank problem working in time $O(n)$ and in sublinear space. They also show how to reduce the order statistics problem to the rank problem by calling $O(\log n)$ instances of the rank problem. This gives an algorithm running in $O(n \log n)$ time. They remark that their solution to the rank problem could run in time $O(n^{5/6+o(1)})$ if it was possible to compute the sum $\sum_{i=1}^{n} \lfloor xi \rfloor$, for a rational x, in this time or faster. This sum is related to counting lattice points in right triangles. A simple algorithm for that task running in logarithmic time can be found in [3]. Nevertheless, for completeness we present a simple logarithmic algorithm computing that sum in Section 3.

The $O(n^{5/6+o(1)})$ solution is complicated. For instance it involves summation of Möbius function and subexpotential integer factorization. In Section 2.3 we

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 218–229, 2007.

present a simple algorithm for the rank problem with time complexity $O(n^{3/4})$ and space complexity $O(\sqrt{n})$. We assume RAM as a model of computation.[1]

What remains is to show a faster reduction in order to find order statistics in sublinear time. In [1] the reduction was made in two stages. The first stage consists of finding out the interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$ containing the k-th term in the sequence. The interval is computed by a binary search with calling the rank problem $O(\log n)$ times. The second stage tracks the searched term by checking all fractions in the interval. Since there are at most n such fractions, in the worst case this stage runs in $O(n)$ time, which dominates time complexity of the reduction. In [1] the authors proposed to take smaller interval $\left[\frac{j}{n^2}, \frac{j+1}{n^2}\right)$, since this interval contains exactly one fraction. However, there is a problem of tracking that fraction, which is not solved by them. In Section 2.2 we show solution to that problem. We also show another, more direct reduction running in logarithmic time and using $O(\log n)$ calls to the rank problem. This reduction is obtained by exploring Stern–Brocot tree in a smart way.

2 Computing Order Statistics in the Farey Sequences

2.1 An $O(n^2)$ Time Algorithm

We show two $O(n^2)$ time methods. The working time follows the number of elements in the sequence $\mathcal{F}_n$, which is asymptotically equal to $\frac{3}{\pi^2}n^2$.

We use the following property of the Farey sequences. For two consecutive fractions $\frac{a}{b} < \frac{c}{d}$ in a Farey sequence, the first fraction that appears between them is $\frac{a+c}{b+d}$ – the mediant of these two fractions. The mediant is already reduced and it firstly appears in $\mathcal{F}_{b+d}$. Using this property one can successively compute all fractions. That way the Stern–Brocot tree[2] is obtained. Farey fractions form a subtree. In–order traversal gives an $O(n^2)$ time and $O(n)$ space algorithm. Space complexity depends on the depth of the Farey tree.

The second $O(n^2)$ method is a straightforward application of a surprising formula. For three consecutive fractions $\frac{a}{b} < \frac{c}{d} < \frac{e}{f}$ the following holds:

$$\frac{e}{f} = \frac{tc - a}{td - b}, \text{ where } t = \left\lfloor \frac{b+n}{d} \right\rfloor.$$

That method works in optimal $O(1)$ space.

2.2 Reduction to the Rank Problem

In order to achieve a solution faster than quadratic one we reduce the problem of finding given term of the Farey sequence to a problem of counting the number of fractions bounded by a real number (the rank problem). To be more precise, for

[1] In RAM (Random Access Machine) single cell can store arbitrarily large integers. Cell access or arithmetic operations on cells are performed in constant time. Also memory complexity is measured in cells.

[2] For a description of Stern–Brocot tree together with its properties we refer to [2].

given positive integer n and real number $x \in [0,1]$ we want to find the number of fractions $\frac{a}{b}$ belonging to sequence $\mathcal{F}_n$ not larger than x. We show how to solve the original problem given an algorithm for the rank problem.

Reduction in linear time. We recall the reduction from [1]. Firstly, the interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$ containing the k-th term is searched by a binary search starting from the interval $\left[\frac{0}{n}, \frac{n}{n}\right)$, splitting interval $\left[\frac{l}{n}, \frac{r}{m}\right)$ into two smaller intervals $\left[\frac{l}{n}, \frac{m}{n}\right)$ and $\left[\frac{m}{n}, \frac{r}{n}\right)$, where $m = \left\lfloor \frac{l+r}{2} \right\rfloor$. Next, we track the fraction in $\left[\frac{j}{n}, \frac{j+1}{n}\right)$. Because the size of the interval is $\frac{1}{n}$, it contains at most one fraction with denominator $b \leq n$. That fraction can be found in constant time since numerator must be $\left\lfloor \frac{(j+1)b-1}{n} \right\rfloor$. We check all such fractions for all possible denominators. Total tracking time is $O(n)$ and whole reduction also works in time $O(n)$.

It is sufficient to use the above reduction to construct roughly linear time solution, but it is not enough if we want to create a sublinear algorithm. Therefore, we need faster reduction. We show two reductions with logarithmic time.

Smaller interval. First, we can tune up the construction using intervals. As it was suggested in [1] we can find smaller interval $\left[\frac{j}{n^2}, \frac{j+1}{n^2}\right)$ also by a binary search. Now in such interval there is at most one fraction in $\mathcal{F}_n$ which belongs to that interval. This is because the size of the interval is $\frac{1}{n^2}$ and because the following inequality holds for every two consecutive fractions $\frac{a}{b} < \frac{c}{d}$ in sequence $\mathcal{F}_n$:

$$\frac{c}{d} - \frac{a}{b} = \frac{1}{bd} \leq \frac{1}{n^2}.$$

The k-th term of the Farey sequence $\mathcal{F}_n$ is the only fraction from $\mathcal{F}_n$ in the interval $\left[\frac{j}{n^2}, \frac{j+1}{n^2}\right)$. What is left is to track this fraction. We use the Stern–Brocot tree to this task. The Stern–Brocot tree allows us to explore all irreducible fraction in an organized way. We start from two fractions $\frac{0}{1}$ and $\frac{1}{0}$. These fractions represent the interval where the k-th term resides. We repeatedly narrow that interval to enclose the interval $\left[\frac{j}{n^2}, \frac{j+1}{n^2}\right)$ until we find the fraction. Assume we have already narrowed the interval to fractions $\frac{a}{b}$ and $\frac{c}{d}$, such that

$$\frac{a}{b} < \frac{j}{n^2} < \frac{j+1}{n^2} \leq \frac{c}{d}.$$

Then, in a single iteration we split the interval by the mediant $\frac{a+c}{b+d}$. If the mediant falls into the interval $\left[\frac{j}{n^2}, \frac{j+1}{n^2}\right)$, then the k-th term is found and this is the mediant $\frac{a+c}{b+d}$. Otherwise, we replace one of the fractions $\frac{a}{b}$ and $\frac{c}{d}$ by $\frac{a+c}{b+d}$. If $\frac{a+c}{b+d} < \frac{j}{n^2}$, we replace fraction $\frac{a}{b}$, otherwise $\frac{a+c}{b+d} \geq \frac{j+1}{n^2}$ and we replace $\frac{c}{d}$.

The above procedure guarantees successful tracking. However, the time complexity is $O(n)$ since in the worst case n iterations are needed. For instance for $k = 1$ we replace the right fraction n times consecutively to $\frac{1}{1}, \frac{1}{2}, \ldots, \frac{1}{n}$. This problem can be solved by grouping successive substitutions of left or right fractions.

Suppose we replace several times the left fraction. After first substitution we have fractions $\frac{a+c}{b+d}$ and $\frac{c}{d}$. After second substitution the fractions become $\frac{a+2c}{b+2d}$

and $\frac{c}{d}$ and so on. Generally, after t substitutions of left fractions the final fraction is equal to $\frac{a+tc}{b+td}$. In the above procedure we replace the left fractions as long as

$$\frac{a+tc}{b+td} < \frac{j}{n^2}. \tag{1}$$

If t is the largest integer satisfying the above inequality then for the next mediant we have $\frac{j}{n^2} \leq \frac{a+(t+1)c}{b+(t+1)d}$. If there is also $\frac{a+(t+1)c}{b+(t+1)d} < \frac{j+1}{n^2}$, then the fraction is found and we can finish the search. Otherwise, the next mediant will substitute the right fraction.

We see that we can make all successive iterations replacing the left fraction at once. We only need to determine the value of t. After rewriting (1) we get

$$(n^2c - jd)t < jb - n^2a. \tag{2}$$

Because $\frac{j}{n^2} < \frac{c}{d}$ we know that $n^2c - jd > 0$, so (2) is equivalent to

$$t < \frac{jb - n^2a}{n^2c - jd}.$$

The largest t satisfying that inequality is

$$t = \left\lceil \frac{jb - n^2a}{n^2c - jd} \right\rceil - 1. \tag{3}$$

Analogously we analyze a situation when we replace several times the right fractions. After t substitutions the right fraction is equal to $\frac{ta+c}{tb+d}$. Replacement takes place as long as

$$\frac{j+1}{n^2} \leq \frac{ta+c}{tb+d}.$$

The largest t satisfying the above inequality is

$$t = \left\lfloor \frac{(j+1)d - n^2c}{n^2a - (j+1)b} \right\rfloor. \tag{4}$$

We conclude that the procedure of tracking the fraction from the interval $\left[\frac{j}{n^2}, \frac{j+1}{n^2}\right)$ can be much faster if we group steps in one direction. In the first iteration we make all steps to the left replacing the right fraction. Then, in the next iteration, we make all steps to the right replacing the left fraction. Next, we make all steps to the left and so on until we find the fraction from the given interval. In a single iteration, if we are going to the right, we replace the left fraction by $\frac{a+tc}{b+td}$ where t is given by (3) and if we are going to the left, we replace the right fraction by $\frac{ta+c}{tb+d}$ where t is given by (4). Excluding the first iteration we know that t is always at least one since the next mediant has to replace the opposite fraction. It means that the denominator is always replaced at least by a sum of the previous two denominators. Therefore, the sequence of successive denominators increases as fast as Fibonacci numbers. Thus, the number of iterations in this procedure is $O(\log n)$.

Exploring Stern–Brocot tree directly. Suppose we are able to solve the rank problem in "reasonable" time. This means that for every fraction $\frac{a}{b} \in \mathcal{F}_n$ we can compare it with the k-th term of the sequence $\mathcal{F}_n$. Using only comparisons we can descend the Stern–Brocot tree down to the searched fraction. We start from the interval $\left(\frac{0}{1}, \frac{1}{0}\right)$. Then, we repeatedly split the interval by the mediant and choose the interval containing the searched fraction. That is, if we have interval $\left(\frac{a}{b}, \frac{c}{d}\right)$, we take the mediant $\frac{a+c}{b+d}$ and compare it with the k-th term. If the number of fractions in $\mathcal{F}_n$ not larger than $\frac{a+c}{b+d}$ equals k, then we get the result; if it is larger than k, then the term lies in the interval $\left(\frac{a}{b}, \frac{a+c}{b+d}\right)$; and if it is less than k, then the term lies in $\left(\frac{a+c}{b+d}, \frac{c}{d}\right)$.

As in the previous reduction in the worst case we have to call the rank problem $O(n)$ times and, as previously, we have to optimize the search by grouping moves in a single direction. However, here we cannot give an explicit formula for the number of steps t, because we can only ask on which side of a given fraction the searched term lies. Fortunately, there is a technique for finding t using at most $O(\log t)$ questions.

The technique can be used when for any integer s we can ask whether $s < t$ or $s = t$ or $s > t$. First, for successive $i = 0, 1, 2, \ldots$ we check whether $2^i < t$. If it shows that $t = 2^i$ for some i, then we find t in $i + 1$ questions. Otherwise, we can find the smallest positive integer l such that $2^{l-1} < t < 2^l$ after $l + 1$ questions. In this case we perform a binary search for t in the interval $(2^{l-1}, 2^l)$. The binary search takes at most $l - 1$ questions so the whole procedure has at most $2l \leq 2 \log t$ questions.

Using the above technique for grouping steps in a single direction we can achieve a method which asks only at most $4 \log n$ questions. We formalize this method for clearer analysis.

Let $\frac{P_0}{Q_0} = \frac{1}{0}$ and $\frac{P_1}{Q_1} = \frac{0}{1}$ so that the starting interval is $\left(\frac{P_1}{Q_1}, \frac{P_0}{Q_0}\right)$. In the i-th iteration, for $i = 2, 3, \ldots$, we construct fraction $\frac{P_i}{Q_i}$. For even i we move to the left and replace the right fraction. For odd i we move to the right and replace the left fraction.

Assume i is even. In that case the interval is $\left(\frac{P_{i-1}}{Q_{i-1}}, \frac{P_{i-2}}{Q_{i-2}}\right)$. Here we are moving to the left adding the left fraction to the right as many times as possible. We search for the largest t_i such that the searched fraction is not larger than

$$\frac{t_i P_{i-1} + P_{i-2}}{t_i Q_{i-1} + Q_{i-2}}.$$

Then we replace the right fraction by it, thus $P_i = t_i P_{i-1} + P_{i-2}$ and $Q_i = t_i Q_{i-1} + Q_{i-2}$. When i is odd we proceed analogously but in opposite direction.

We repeat calculating successive $\frac{P_i}{Q_i}$ until for some i that fraction is the k-term of the sequence $\mathcal{F}_n$. The above procedure is nothing new. In fact it has strict connection with continued fractions. One may prove that

$$\frac{P_i}{Q_i} = \cfrac{1}{t_2 + \cfrac{1}{\quad\vdots\quad \cfrac{}{t_{i-1} + \cfrac{1}{t_i}}}}$$

Let us analyze the time complexity. First, observe that every t_i is positive. For each $i = 2, 3, \ldots$ we ask at most $2l_i$ questions in the i-th iteration, where $2^{l_i - 1} \leq t_i < 2^{l_i}$. Suppose we made h iterations so the searched fraction is $\frac{P_h}{Q_h}$ and $Q_h \leq n$. From the recursive formula for $Q_i = t_i Q_{i-1} + Q_{i-2}$ we conclude that sequence Q_i is non-decreasing and thus the following inequality holds $Q_i \geq (t_i + 1)Q_{i-1}$. Hence,

$$\begin{aligned} n \geq Q_h &\geq (t_h + 1)Q_{h-1} \geq (t_h + 1)(t_{h-1} + 1)Q_{h-2} \geq \cdots \\ &\geq (t_h + 1) \cdots (t_2 + 1)Q_1 = (t_h + 1) \cdots (t_2 + 1). \end{aligned}$$

There are two consequences of the above inequality. First, since $t_i + 1 \geq 2$ we have $n \geq 2^{h-1}$ so $h - 1 \leq \log n$. Second, since $t_i \geq 2^{l_i - 1}$ we have $n \geq 2^{l_h + \ldots + l_2 - (h-1)}$. Combining it we get

$$l_2 + \ldots + l_h \leq \log n + (h - 1) \leq 2 \log n.$$

Therefore, the total number of questions is $O(\log n)$, since in all iterations we ask at most $2(l_2 + \ldots + l_h)$ questions.

2.3 Solution to the Rank Problem

Let $S_n(x)$ be the number we are searching for, i.e. the number of irreducible fractions $\frac{a}{b}$ such that $\frac{a}{b} \leq x$ and $b \leq n$. For simplicity we will sometimes write S_n instead of $S_n(x)$ because in fact x is fixed.

Playing with symbol $S_n(x)$ and grouping by gcd we can get a recursive formula, which will be a starting point to our algorithm:

$$\begin{aligned} S_n(x) &= \left| \left\{ \frac{a}{b} \mid b \leq n \wedge \frac{a}{b} \leq x \wedge \gcd(a, b) = 1 \right\} \right| \\ &= \left| \left\{ \frac{a}{b} \mid b \leq n \wedge \frac{a}{b} \leq x \right\} \right| - \sum_{d \geq 2} \left| \left\{ \frac{a}{b} \mid b \leq n \wedge \frac{a}{b} \leq x \wedge \gcd(a, b) = d \right\} \right| \\ &= \sum_{b=1}^{n} \lfloor bx \rfloor - \sum_{d \geq 2} S_{\lfloor \frac{n}{d} \rfloor}(x). \end{aligned}$$

We explain each step. For given constraints the number of irreducible fractions is the total number of fractions minus the number of fractions with gcd of numerator and denominator equal or larger 2. It is written in the second line of the equation. The number of fractions with given denominator b less than or equal to x is $\lfloor bx \rfloor$, so the number of all fractions less than or equal to x is the sum:

$$\sum_{b=1}^{n} \lfloor bx \rfloor.$$

We should also explain equality

$$\left|\left\{\frac{a}{b} \mid b \le n \wedge \frac{a}{b} \le x \wedge \gcd(a,b) = d\right\}\right| = S_{\lfloor \frac{n}{d} \rfloor}(x).$$

Every fraction $\frac{a}{b}$ with $\gcd(a,b) = d$ has form $\frac{a'd}{b'd}$, where $a'd = a$, $b'd = b$ and $\gcd(a',b') = 1$. It means that fraction $\frac{a'}{b'}$ is irreducible and $b' \le \frac{n}{d}$, since $b \le n$. The number of irreducible fractions such that $\frac{a'}{b'} = \frac{a}{b} \le x$ is exactly $S_{\lfloor \frac{n}{d} \rfloor}(x)$.

Let us look again at the recursive formula:

$$S_n = \sum_{b=1}^{n} \lfloor bx \rfloor - \sum_{d \ge 2} S_{\lfloor \frac{n}{d} \rfloor}. \tag{5}$$

In fact, x is always a rational number in our algorithm. In that case the sum $\sum_{b=1}^{n} \lfloor bx \rfloor$ can be calculated in $O(\text{polylog}(n))$. It is shown in Section 3. So the only problem is to calculate the sum $\sum_{d \ge 2} S_{\lfloor \frac{n}{d} \rfloor}$. Let us focus on how many different summands there are. For $d \le \sqrt{n}$ all expressions $S_{\lfloor \frac{n}{d} \rfloor}(x)$ are unique. If $d > \sqrt{n}$, then $\frac{n}{d} < \sqrt{n}$, so for $d > \sqrt{n}$ there are at most $\sqrt{n}$ different summands. Therefore, on the right hand side of the formula (5) there are $O(\sqrt{n})$ summands.

Moreover, in deeper level of recursion, occurrence of symbol S_i is only possible if $i = \lfloor \frac{n}{d} \rfloor$, for some positive integer d. This property follows from the equality

$$\left\lfloor \frac{\lfloor \frac{n}{d_1} \rfloor}{d_2} \right\rfloor = \left\lfloor \frac{n}{d_1 d_2} \right\rfloor.$$

We are left with computing S_i where i is from set $I = \left\{ \lfloor \frac{n}{d} \rfloor \mid d \ge 1 \right\}$. We split the set to two sets $I_1 = \{1, 2, \ldots, \lfloor \sqrt{n} \rfloor\}$ and $I_2 = \left\{ \left\lfloor \frac{n}{\lfloor \sqrt{n} \rfloor} \right\rfloor, \ldots, \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{1} \rfloor \right\}$. Each of these sets has $\lfloor \sqrt{n} \rfloor$ elements and they sum to I. We use dynamic programming to calculate successive S_i for increasing $i \in I$. To calculate S_i we use formula

$$S_i = \sum_{b=1}^{i} \lfloor bx \rfloor - \sum_{d \ge 2} S_{\lfloor \frac{i}{d} \rfloor}. \tag{6}$$

As it was already mentioned actual size of the sum $\sum_{d \ge 2} S_{\lfloor \frac{i}{d} \rfloor}$ is $O(\sqrt{i})$. For each symbol S_j occurring in the sum we can find its multiplicity in constant time, simply by finding interval for d values for which $\lfloor \frac{n}{d} \rfloor = j$. Therefore, the time complexity of calculating the right hand side of (6) is $O(\sqrt{i})$. The memory complexity is $O(\sqrt{n})$, since we have to store only S_i for $i \in I$.

Surprisingly, the above algorithm for calculating all S_i works in $O(n^{3/4})$. We prove it in two parts. In the first part let us determine the time of calculating S_i for all $i \in I_1$:

$$\sum_{1 \le i \le \sqrt{n}} O(\sqrt{i}) \subseteq \sum_{1 \le i \le \sqrt{n}} O\left(\sqrt{\sqrt{n}}\right) = O\left(\sqrt{n} \cdot \sqrt{\sqrt{n}}\right) = O(n^{3/4}).$$

For the second part observe that $I_2 = \left\{ \left\lfloor \frac{n}{d} \right\rfloor \mid 1 \le d \le \sqrt{n} \right\}$. Thus, the time complexity of calculating S_i for all $i \in I_2$ is

$$\sum_{1 \le d \le \sqrt{n}} O\left(\sqrt{\left\lfloor \frac{n}{d} \right\rfloor}\right) = O\left(\sqrt{n} \sum_{1 \le d \le \sqrt{n}} \frac{1}{\sqrt{d}}\right).$$

Using the asymptotic equality

$$\sum_{1 \le i \le x} \frac{1}{\sqrt{i}} = O(\sqrt{x})$$

we get the result

$$O\left(\sqrt{n} \cdot \sqrt{\sqrt{n}}\right) = O(n^{3/4}).$$

3 Computing $\sum_{i=1}^{n} \left\lfloor \frac{a}{b} i \right\rfloor$

In this section we present a simple algorithm for computing the sum $\sum_{i=1}^{n} \lfloor \frac{a}{b} i \rfloor$, where $\frac{a}{b}$ is a non-negative irreducible fraction. We remark that a polynomial time algorithm (in size of a, b and n) was previously presented in [4] or in [5]. However in these papers used methods are rather complicated for easy implementation. Much simpler algorithm for computing lattice points in a rational right triangle was presented in [3]. Although our algorithm is very similar, we decided to include a description for this specific task for two reasons. First, since a solution to the rank problem needs to calculate this sum, we want to make the whole procedure to be completed. Second, the sum is a special case of rational right triangle. Therefore, formulas used in an algorithm can be easier determined and slightly simplified comparing to formulas presented in [3].

A graphical representation of the given sum is shown in Fig. 1. The value of the sum is the number of lattice points in a triangle bounded by X–axis and lines $x = n$ and $y = \frac{a}{b}x$ excluding lattice points on X–axis. That representation will help to see properties of the sum.

Let us denote

$$T(n, a, b) = \sum_{i=1}^{n} \left\lfloor \frac{a}{b} i \right\rfloor.$$

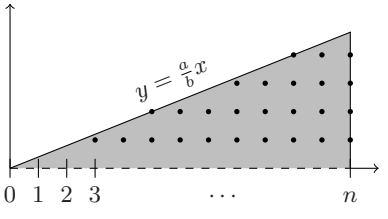


Fig. 1. Graphical representation of the sum $\sum_{i=1}^{n} \lfloor \frac{a}{b} i \rfloor$

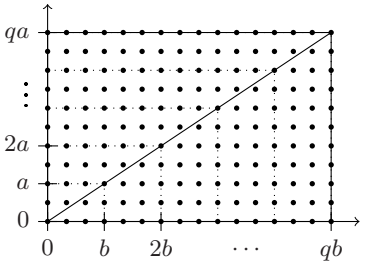


Fig. 2. Case when n is divisible by b

We develop recursive formulas for $T(n, a, b)$. These formulas lead to a straightforward polynomial time algorithm (in size of n, a, b).

3.1 Case $n \geq b$

If n is divisible by b, we can derive a closed form. Let $n = qb$ and look at Fig. 2. We can easily calculate the number of lattice points in the lower right triangle. Observe that the lower right triangle is identical to the upper left triangle and it contains the same number of lattice points. Summing up both triangles we get the rectangle with diagonal counted twice. The number of lattice points in the rectangle is $(qb+1)(qa+1)$ and the number of points on the diagonal is equal to $q+1$. The sum of both values divided by two gives the number of lattice points in a triangle. Now, we subtract the number of lattice points on X-axis getting the result:

$$T(qb, a, b) = \frac{(qa+1)(qb+1)+q+1}{2} - (qb+1) = \frac{q(qab-b+a+1)}{2}. \tag{7}$$

More generally, suppose $n \geq b$ and let $n = qb + r$, where $q \geq 1$ and $0 \leq r < b$. The sum can be splitted into three parts:

$$\sum_{i=1}^{qb+r} \left\lfloor \frac{a}{b} i \right\rfloor = \sum_{i=1}^{qb} \left\lfloor \frac{a}{b} i \right\rfloor + \sum_{i=qb+1}^{qb+r} \left\lfloor \frac{a}{b} i \right\rfloor = \sum_{i=1}^{qb} \left\lfloor \frac{a}{b} i \right\rfloor + \sum_{i=1}^{r} \left\lfloor \frac{a}{b} (qb+i) \right\rfloor$$
$$= \sum_{i=1}^{qb} \left\lfloor \frac{a}{b} i \right\rfloor + r \cdot aq + \sum_{i=1}^{r} \left\lfloor \frac{a}{b} i \right\rfloor.$$

See Fig. 3 for intuition. As a result we get the equation:

$$T(qb + r, a, b) = T(qb, a, b) + rqa + T(r, a, b). \tag{8}$$

As a consequence of the above formula together with equation (7) we can reduce n below b in a single step. Therefore, in the succeeding sections we assume that $n < b$. Notice that it also means that there is no integral point on the line $y = \frac{a}{b}x$ for $x = 1, 2, \ldots, n$.

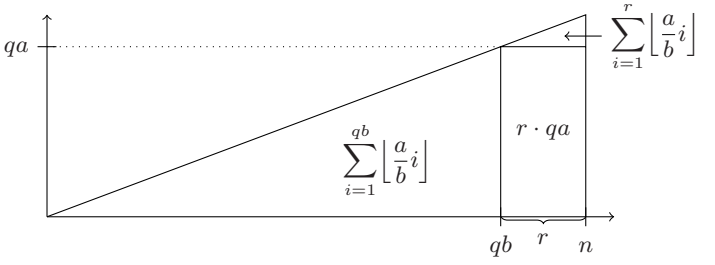


Fig. 3. Case $n \geq b$

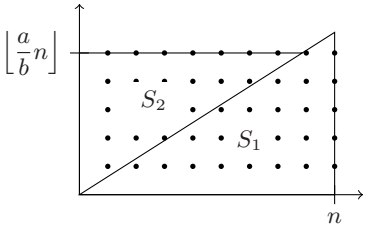


Fig. 4. Graphical representation of sums $\sum \lfloor \frac{a}{b} i \rfloor$ and $\sum \lfloor \frac{b}{a} i \rfloor$

3.2 Case $a \geq b$

If $a = qb + r$ for some $q \geq 1$ and $0 \leq r < b$, we can rewrite:

$$\sum_{i=1}^{n} \left\lfloor \frac{a}{b} i \right\rfloor = \sum_{i=1}^{n} \left\lfloor \frac{qb+r}{b} i \right\rfloor = \sum_{i=1}^{n} qi + \sum_{i=1}^{n} \left\lfloor \frac{r}{b} i \right\rfloor = q\frac{n(n+1)}{2} + \sum_{i=1}^{n} \left\lfloor \frac{r}{b} i \right\rfloor.$$

Thus in this case we have the formula:

$$T(n, qb + r, b) = \frac{n(n+1)}{2} q + T(n, r, b). \tag{9}$$

3.3 Inverting $\frac{a}{b}$

We use graphical representation to correlate sums $\sum \lfloor \frac{a}{b} i \rfloor$ and $\sum \lfloor \frac{b}{a} i \rfloor$ in one equation. In Fig. 4 area labelled S_1 represents the sum $\sum_{i=1}^{n} \lfloor \frac{a}{b} i \rfloor$. The largest x and y coordinates of lattice points in this area are n and $\lfloor \frac{a}{b} n \rfloor$ respectively. Consider the rectangular set R of lattice points with x coordinates spanning from 1 to n and with y coordinates spanning from 1 to $\lfloor \frac{a}{b} n \rfloor$. This set has size $n \lfloor \frac{a}{b} n \rfloor$. Let S_2 be complement of S_1 in R. We assumed that $n < b$ and there is no element in R lying on the line $y = \frac{a}{b} x$. Therefore, for given $j = 1, \ldots, \lfloor \frac{a}{b} n \rfloor$, the number of lattice points with y coordinate set to j in area S_2 is equal to $\lfloor \frac{b}{a} j \rfloor$. Hence, the size of S_2 is $\sum_{j=1}^{\lfloor \frac{a}{b} n \rfloor} \lfloor \frac{b}{a} j \rfloor$. Since $|S_1| + |S_2| = |R|$ we have

$$\sum_{i=1}^{n} \left\lfloor \frac{a}{b} i \right\rfloor + \sum_{j=1}^{\lfloor \frac{a}{b} n \rfloor} \left\lfloor \frac{b}{a} j \right\rfloor = n \left\lfloor \frac{a}{b} n \right\rfloor.$$

Thus, the last recursive formula is

$$T(n,a,b) = n\left\lfloor \frac{a}{b}n \right\rfloor - T\left(\left\lfloor \frac{a}{b}n \right\rfloor, b, a\right). \tag{10}$$

It allows to swap a with b in $T(\cdot,a,b)$. It can be used to make $a \geq b$. Notice that after swapping a with b our assumption that $n < b$ holds since if $n < b$, then $\frac{a}{b}n < a$ and $\lfloor \frac{a}{b}n \rfloor < a$.

3.4 Final Algorithm

Combining presented recursive formulas for $T(n,a,b)$ we can design the final algorithm. The procedure is similar to the Euclidean algorithm. First, if $n \geq b$ reduce n using (8) making $n < b$, then repeat the following steps until n or a reaches zero. If $a < b$, use (10) to exchange a with b. Next, use (9) to reduce a to $a \bmod b$.

The number of steps in the above procedure is $O(\log \max(n,a,b))$ as it is in the Euclidean algorithm. The algorithm is fairly simple and it can be written in the recursive fashion.

4 Summary and Remarks

We presented a simple sublinear algorithm for the rank problem. We showed that this algorithm has $O(n^{3/4})$ time complexity and needs $O(\sqrt{n}\,)$ space.

The order statistics problem was reduced to the rank problem. We included two reductions. Both call the rank problem $O(\log n)$ times and run in $O(\log n)$ time. Therefore, we showed that the order statistics in the Farey sequences can be computed in $O(n^{3/4} \log n)$ time.

In reduction exploring the Brocot–Stern tree, we showed how to find a rational number if we are only allowed to compare it with fractions. We remark that this technique can be used in other fields. For instance, it can be used to expand a real number into a continued fraction. We do not need the value of this number. We only need a comparison procedure between that number and an arbitrary fraction. For instance, numbers with such property are algebraic numbers. However, in this case there are other methods of expanding them into continued fractions [6]. The usefullness of the presented technique should be further investigated.

References

1. Pătrascu, C.E, Pătrascu, M.: Computing order statistics in the Farey sequence. In: Buell, D.A. (ed.) Algorithmic Number Theory. LNCS, vol. 3076, pp. 358–366. Springer, Heidelberg (2004)
2. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics, 2nd edn. Addison-Wesley, London, UK (1994)

3. Yanagisawa, H.: A simple algorithm for lattice point counting in rational polygons. Research report, IBM Research, Tokyo Research Laboratory (August 2005)
4. Barvinok, A.I: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Mathematics of Operations Research 19(4), 769–779 (1994)
5. Beck, M., Robins, S.: Explicit and efficient formulas for the lattice point count in rational polygons using Dedekind–Rademacher sums. Discrete and Computational Geometry 27(4), 443–459 (2002)
6. Brent, R.P., van der Poorten, A.J., te Riele, H.: A comparative study of algorithms for computing continued fractions of algebraic numbers. In: Cohen, H. (ed.) Algorithmic Number Theory. LNCS, vol. 1122, pp. 35–47. Springer, Heidelberg (1996)

New Results on Minimax Regret Single Facility Ordered Median Location Problems on Networks*

Justo Puerto, Antonio M. Rodriguez-Chia, and Arie Tamir

[1] Facultad de Matemáticas. Universidad de Sevilla
[2] Facultad de Ciencias. Universidad de Cádiz
[3] School of Mathematical Sciences. Tel Aviv University

Abstract. We consider the single facility ordered median location problem with uncertainty in the parameters (weights) defining the objective function. We study two cases. In the first case the uncertain weights belong to a region with a finite number of extreme points, and in the second case they must also satisfy some order constraints and belong to some box, (convex case). To deal with the uncertainty we apply the minimax regret approach, providing strongly polynomial time algorithms to solve these problems.

Keywords: Analysis of algorithms, networks, facility location.

1 Introduction

The definition of an instance of an optimization problem requires the specification of the problem parameters, like resource limitations and coefficients of the objective function in a linear program, or edge capacities in network flow problems, which may be uncertain or imprecise. Uncertainty/imprecision can be structured through the concept of a scenario which corresponds to an assignment of plausible values to the model parameters. In general, the set of all admissible scenarios may depend on the properties of the underlying model, and on some possible known relationships between the model parameters. Nevertheless, we note that in most published studies it has been assumed that each parameter can independently take on values in some prespecified interval. This is the so called interval data approach.

One of the most common approaches to deal with uncertain data is throughout the minimax absolute regret criterion. In this approach the goal is to minimize the worst case opportunity loss, defined as the difference between the achieved objective-function value and the optimal objective-function value under the realized scenario. We refer the reader to [2,3,5,8,12,17], where recent results on this subject for general optimization problems are described.

* Partially supported by grants n. MTM2004-0909, SAB2005-0095, P06-BFM-01366, MTM2007-67433-C02.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 230–240, 2007.

We were motivated by facility location optimization problems where the exact nature of the optimality criteria was uncertain. Consider, for example, a location model in which the central administration subsidizes the transportation cost of the users only after the establishment of a server. Hence, at the moment when the facility has to be established, it is not yet clear what is the cost function that the central administration will apply for determining the magnitude of the subsidy. The administration may decide that the subsidy will be proportional to the distances traveled to the facility by all the customers with the exception of some unknown subset of outliers, e.g., those customers who are too close or too far to the server. The uncertainty is in the size or the exact definition of the set of outliers. In such a case, the criterion to be chosen 'a priori' for the location of the server might be taken as minimizing the regret of the decision among a certain family of criteria.

The approach that we suggest to deal with this uncertainty is to limit ourselves to certain families of objective functions, and apply the approach of minimizing the maximum regret with respect to the selection of an objective within the pre-specified. Our approach is different from the approach in the existing literature on minimax regret facility location models, where the objective function used is usually assumed to be known and certain ([4,5]). Specifically, in this paper, we will restrict ourselves to the family of ordered median functions (OMF) which has been studied extensively in the last decade in location theory, see [9,15]. This family unifies large variety of criteria used in location modeling.

The OMF is a real function defined on $\mathbb{R}^n$ and characterized by a sequence of reals, $\lambda = (\lambda_1, ..., \lambda_n)$. For a given point $z \in \mathbb{R}^n$, let $\bar{z} \in \mathbb{R}^n$ be the vector obtained from z by sorting its components in nondecreasing order. In the context of a single facility (server) location model with n demand points, the OMF objective is applied as follows. Let x denote the location of the server in the respective metric space, and let $z(x)$ denote the vector of the n weighted distances of the demand points to the server at x. The value of the ordered median objective at x is then defined as the scalar product of λ with $\bar{z}(x)$. As noted above, this function unifies and generalizes the classical and most common criteria, i.e., center and median, used in location modeling. (We get the median objective when $\lambda_i = 1$, $i = 1, ..., n$, and the center objective when $\lambda_i = 0$, $i = 1, ..., n-1$ and $\lambda_n = 1$.) Another important case is the k-centrum objective, where the goal is to minimize the sum of the k-largest weighted distances to the server. This case is characterized by $\lambda_i = 0$, $i = 1, ..., n-k$, and $\lambda_i = 1$, $i = n-k+1, ..., n$. In addition to the above examples, the ordered median objective generalizes other popular criteria often used in facility location studies, e.g., centdian and (k_1, k_2)-trimmed mean.

In this paper, we solve a variety of single facility minimax regret ordered median problems on general networks finding best solutions on each edge. The reader can find further details in [16]. A summary of the results is given in Table 1.

The paper is organized as follows. In Section 2, we present the single facility ordered median problem in general networks. Section 3 introduces the minimax

Table 1. Summary of results

Objective function	Complexity
OMF with lower and upper bounds	$O(m^2n^4 \log n)$
(k_1, k_2)-trimmed mean	$O(mn^4)$
k-centrum	$O(mn^2 \log^2 n)$
Convex OMF with lower and upper bounds	$O(m^2n^6 \log^4 n)$
Convex OMF with lower and upper bounds on trees	$O(n^6 \log^4 n)$

regret ordered median problem and some results concerning the convexity of the objective function. In Section 4, we develop strongly polynomial algorithms for this type of problems when the feasible region of the λ-weights has a finite number of extreme points. Section 5 is devoted to analyze the convex case, which is defined by the property that the λ-weights are given in nondecreasing order; for this case a strongly polynomial algorithm is developed.

2 Notation

Let $G = (V, E)$ be an undirected graph with node set $V = \{v_1, ..., v_n\}$ and edge set E, $|E| = m$. Each edge $e \in E$, has a positive length l_e, and is assumed to be rectifiable. In particular, an edge e is identified as an interval of length l_e, so that we can refer to its interior points. Let $\mathcal{A}(G)$ denote the continuum set of points on the edges of G. Each subgraph of G is also viewed as a subset of $\mathcal{A}(G)$, e.g., each edge $e \in E$ is a subset of $\mathcal{A}(G)$. We refer to an interior point on an edge by its distance along the edge to the nodes of the edge.

The edge lengths induce a distance function d on $\mathcal{A}(G)$ and thus $\mathcal{A}(G)$ is a metric space , see [18]. We consider a set of nonnegative weights $\{w_1, \ldots, w_n\}$, called w-weights, where w_i, $i = 1, \ldots, n$, is associated with node v_i and represents the intensity of the demand at this node.

For any $x \in \mathcal{A}(G)$, let $\sigma = \sigma(x)$ be a permutation of the set $\{1, \ldots, n\}$ satisfying $w_{\sigma_1} d(v_{\sigma_1}, x) \leq \ldots \leq w_{\sigma_n} d(v_{\sigma_n}, x)$. For $i = 1, \ldots, n$, denote $d_{(i)}(x) = w_{\sigma_i} d(v_{\sigma_i}, x)$. ($d_{(i)}(x)$ is an i-th smallest element in the set $\{w_j d(v_j, x)\}_j$).

For a given vector $\lambda = (\lambda_1, \ldots, \lambda_n)$ with real components, called λ-weights, *the ordered median function* on $\mathcal{A}(G)$ is defined as

$$f_\lambda(x) := \sum_{i=1}^{n} \lambda_i w_{\sigma_i} d(v_{\sigma_i}, x).$$

The single facility ordered median problem is to minimize $f_\lambda(x)$ over $\mathcal{A}(G)$, [15].

An ordered median function is called convex if $0 \leq \lambda_1 \leq \ldots \leq \lambda_n$.

A point $x \in \mathcal{A}(G)$ is an *equilibrium point* with respect to a pair of nodes v_k, v_l, $k \neq l$, if $w_k\, d(v_k, x) = w_l\, d(v_l, x)$. A point $x \in \mathcal{A}(G)$ is a *bottleneck point* if for some $i = 1, \ldots, n$, with $w_i > 0$, and $e \in E$, x is the unique maximum point of the (concave) function $w_i d(v_i, y)$ when y is restricted to be in e. Denote by $\mathcal{EQ}$ the set of all equilibrium and bottleneck points. This set can be a continuum set

even in the case of a tree network when two nodes are equally weighted. Let EQ be the set consisting of the nodes of G and all the boundary points of $\mathcal{EQ}$. Let B be the subset consisting of the nodes of G and the bottleneck points in EQ. For each $e \in E$ define $EQ(e) = EQ \cap e$ and $B(e) = B \cap e$. Note that $|EQ(e)| = O(n^2)$ and $|B(e)| = O(n)$.

For each $e \in E$ the points in $EQ(e)$ $(B(e))$ induce a partition of the edge e into $|EQ(e)| - 1$ $(|B(e)| - 1)$ subedges. These subedges (subinterals) are viewed as consecutive subintervals of the rectified edge (interval) e.

After computing the distances between all the nodes, the effort to compute and sort the points in $EQ(e)$ $(B(e))$ for each $e \in E$ is $O(n^2 + |EQ(e)| \log n)$ $(O(n + |B(e)| \log n))$. Note that for each subinterval in the partition induced by $B(e)$, each function $w_i d(v_i, x)$ is linear. Moreover, for each open subinterval in the partition induced by $EQ(e)$, no pair of functions in the collection $\{w_i d(v_i, x)\}_i$ intersect.

3 The Minimax Regret Ordered Median Problem

We assume that the vector λ is unknown and can take on any value in some compact set $\Lambda \subset \mathbb{R}^n$. Any $\lambda \in \Lambda$ is called a *scenario* and represents a possible $\lambda-$weight instance. The minimax regret ordered median optimization problem is formally defined by:

$$\min_{x \in \mathcal{A}(G)} R(x) := \max_{\lambda \in \Lambda} \max_{y \in \mathcal{A}(G)} \Big(f_\lambda(x) - f_\lambda(y) \Big).$$

For any choice of the λ-weights, EQ contains at least one optimal solution for the respective ordered median problem, [15]. Thus, for a given $x \in \mathcal{A}(G)$

$$\max_{y \in \mathcal{A}(G)} \Big(f_\lambda(x) - f_\lambda(y) \Big) = \max_{y \in EQ} \Big(f_\lambda(x) - f_\lambda(y) \Big) = f_\lambda(x) - f_\lambda(y^*(\lambda)),$$

where $y^*(\lambda) \in EQ$ is a minimizer of $f_\lambda(u)$ with $u \in \mathcal{A}(G)$. For each fixed $y \in \mathcal{A}(G)$ define

$$R_y(x) = \max_{\lambda \in \Lambda} \Big(f_\lambda(x) - f_\lambda(y) \Big). \tag{1}$$

By definition, for a given pair $x, y \in A(G)$ the function $f_\lambda(x) - f_\lambda(y)$ is linear in λ. Therefore, $R_y(x) = \max_{\lambda \in ext(CH(\Lambda))} \Big(f_\lambda(x) - f_\lambda(y) \Big)$, where $CH(\Lambda)$ is the convex hull of Λ and $ext(CH(\Lambda))$ is the set of extreme points of $CH(\Lambda)$.

With this notation

$$R(x) = \max_{y \in EQ} R_y(x). \tag{2}$$

Consider an edge $e \in E$ and let $x_1^e < \ldots < x_{q(e)}^e$, be the sequence of equilibrium points in $EQ(e)$. $(q(e) = |EQ(e)|.)$ Similarly, let $\bar{x}_1^e < \ldots < \bar{x}_{b(e)}^e$, be the sequence of bottleneck points in $B(e)$. $(b(e) = |B(e)|.)$ (See Figure 1). From the definition of equilibrium and bottleneck points and the above notation, we clearly have the following results.

Lemma 1. *Consider a subedge* $[x_k^e, x_{k+1}^e]$, $1 \leq k < q(e)$. *For any* $\lambda \in \Lambda$, *the function* $f_\lambda(x)$ *is linear on the subedge. For any* $y \in EQ$ *the function* $R_y(x)$ *is continuous and convex on the subedge. Moreover, if the number of extreme points of* $CH(\Lambda)$ *is finite* $R_y(x)$ *is also piecewise linear on the subedge.*

Lemma 2. *For any* $y \in EQ$, *the function* $f_\lambda(y)$ *is linear in* λ. *Let* $P \subseteq \mathcal{A}(G)$ *be a path such that* $f_\lambda(x)$ *is convex on* P *for any* $\lambda \in \Lambda$. *Then for any* $y \in EQ$ *the function* $R_y(x)$ *is convex on* P. *Moreover, the function* $R(x)$ *is convex on* P.

To solve the minimax regret ordered median problem on a general network we will find the best local solution on each edge, i.e., we will solve m subproblems. We refer to each local subproblem as a restricted subproblem.

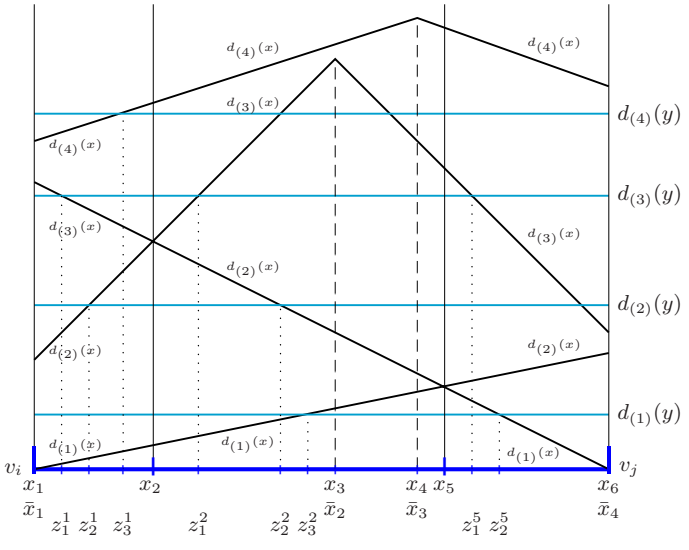


Fig. 1. Equilibrium and bottleneck points

4 Specific Models

In this section, we focus on solving restricted subproblems for a variety of sets Λ, where the number of extreme points of the convex hull of Λ is finite. As noted above, we concentrate on finding the best solution on each edge. Hence, we focus on optimizing $R(x)$ on a given edge e.

We start with the case where $\Lambda = \{(\lambda_1, \ldots, \lambda_n) : a_i \leq \lambda_i \leq b_i,\ i = 1, \ldots, n\}$. The reader may notice that this type of sets is the most common one used in the literature on regret analysis, see [12]. Here, we can strengthen the result in Lemma 1. Consider an edge $e \in E$.

Lemma 3. *For each* $1 \leq k < q(e)$, *and any* $y \in EQ$ *the function* $R_y(x)$ *is the maximum of* n *linear functions in* $[x_k^e, x_{k+1}^e]$.

Lemma 4. *For each $1 \leq k < q(e)$, the function $R(x)$ is the upper envelope of $O(n|EQ|)$ linear functions for all $x \in [x^e_k, x^e_{k+1}]$.*

To solve the restricted problem on an edge e, we first do some preprocessing on this edge. Assume that we have already computed and sorted the equilibrium points in $EQ(e)$. For each triplet $y \in EQ$, $v_i, v_j \in V$, we compute the at most two roots of the equation $w_i d(x, v_i) = w_j d(y, v_j)$ on e. Define $BP(e)$ to be the set consisting of $EQ(e)$ and all these roots. $|BP(e)| = O(n^2|EQ|)$. Moreover, for each i let $z^k_i(y)$ be the solution, if it exists, to the equation $d_{(i)}(x) = d_{(i)}(y)$ (in the variable x), in the interval $[x^e_k, x^e_{k+1}]$. Each point $z^k_i(y)$, is in $BP(e)$.) Finally, we sort the elements in $BP(e)$. The total preprocessing effort is $O(n^2|EQ| \log n)$.

Corollary 1. *After spending $O(n^2|EQ| \log n)$ time on preprocessing, for each $1 \leq k < q(e)$ the local minimizer of $R(x)$ over the interval $[x^e_k, x^e_{k+1}]$ can be computed in $O(n|EQ|)$ time. The optimal solution to the minimax regret ordered median problem on the edge e can be computed in $O(n|EQ||EQ(e)|)$ time.*

The above result implies that for a general network the total time to solve the minimax regret ordered median problem over a box is $O(m^2n^5)$. The latter bound can be further improved. Focusing on a given edge, we dynamically maintain [11] the upper envelope of $O(|EQ|)$ linear functions which define the function $R(x)$ over a refined subinterval defined by two consecutive elements in the set $BP(e)$. Specifically, following the ordering of the elements in $BP(e)$ we update this envelope. For a given element $u \in BP(e)$, if $u \in EQ(e)$ we may need to update $O(|EQ|)$ linear functions since the ordering or some of the slopes of the functions $\{d_{(i)}(x)\}$ change. (For each $y \in EQ$ we need to update at most two elements, per pair of indices j, k such that $w_j d(v_j, u) = w_k d(v_k, u)$, in the sequence $\{d_{(i)}(x) - d_{(i)}(y)\}_i$, or change the slope of a $d_{(i)}(x)$ function, per each j such that u is the maximum of the function $w_j d(v_j, x)$ on e.) If u coincides with some element $z^k_i(y)$ defined above, we need to update one function, per each $y \in EQ$ and j, k such that $w_j d(v_j, u) = d_{(i)}(u) = d_{(i)}(y) = w_k d(v_k, y)$, in the collection. Thus, the total number of insertions and deletions of functions to the collection of $O(|EQ|)$ functions in the upper envelope is $O(|EQ|n^2)$. Using the data structure in Hershberger and Suri [11], each insertion and deletion can be performed in $O(\log n)$ time. Also, the minimum of $R(x)$ over each subinterval connecting two consecutive elements of $BP(e)$ can be computed in $O(\log n)$ time.

Since there are $O(n^2)$ points in $EQ(e)$ and $O(n^2|EQ|)$ points in $BP(e)$ the overall effort to find the best solution on e is $O(n^2|EQ| \log n)$.

Theorem 1. *The total time to solve the single facility minimax regret ordered median problem over a box on a general graph is $O(m^2n^4 \log n)$.*

We note that in some important cases the number of extreme points of the convex hull of Λ is relatively small. Hence, it may be advantageous to consider these extreme points explicitly. This is the case of the family of (k_1, k_2)-trimmed mean functions [15], mentioned in the introduction. (Other cases are analyzed in the next section.) For this family $\Lambda = \{(\lambda_1, \ldots, \lambda_n) : \exists k_1, k_2; k_1 + k_2 < n, \lambda_1 = \ldots =$

$\lambda_{k_1} = \lambda_{n-k_2+1} = \ldots = \lambda_n = 0,\ \lambda_{k_1+1} = \ldots = \lambda_{n-k_2} = 1\}$. The extreme points of the convex hull of Λ are the vectors of the form $(0,\ldots,0,1,\ \ldots,1,0,\ldots,0)$. Therefore, in total there are $O(n^2)$ extreme points.

Consider an edge $e \in E$. We claim that for this family, on each interval defined by two consecutive points of $EQ(e)$, the function $R(x)$ can be described as an upper envelope of $O(n^2)$ linear functions. To facilitate the discussion, for each $k = 1,\ldots,n$, let $S_k(x) = \sum_{i=n-k+1}^{n} d_{(i)}(x)$.

For each $1 \leq s < q(e)$, and for each $x \in [x_s^e, x_{s+1}^e]$, we have $R(x) = \max\limits_{k_1,k_2;k_1+k_2<n} \left(\left(\sum\limits_{i=k_1+1}^{n-k_2} d_{(i)}(x) \right) - \varrho_{k_1,k_2} \right)$, where $\varrho_{k_1,k_2} = \min\limits_{y\in EQ} \sum\limits_{l=k_1+1}^{n-k_2} d_{(l)}(y)$. Hence, $R(x) = \max\limits_{k_1,k_2;k_1+k_2<n} \Big(S_{n-k_1}(x) - S_{k_2}(x) - \min\limits_{y\in EQ} \Big(S_{n-k_1}(y) - S_{k_2}(y) \Big) \Big)$.

Since $R(x)$ is the upper envelope of $O(n^2)$ linear functions, its minimum on the interval $[x_s^e, x_{s+1}^e]$ can be computed in $O(n^2)$ time using the algorithm in [14]. Hence, the solution to the minimax regret ordered median problem on a given edge e for this family of functions, can be obtained in $O(n^2|EQ(e)|)$ time.

Theorem 2. *The total time to solve the single facility minimax regret (k_1, k_2)-trimmed mean problem on a general graph is $O(mn^4)$.*

5 Minimax Regret Convex Ordered Median Problem

In this section we analyze the minimax regret convex ordered median problem which includes several interesting and most common families of functions used in Location Theory.

The first is the family of k-centrum functions where k can vary between 1 and n (see [19]). We have $\Lambda = \{(\lambda_1,\ldots,\lambda_n) : \lambda_1 \leq \ldots \leq \lambda_n \text{ and } \lambda_i \in \{0,1\},\ i = 1,...,n\}$. The n extreme points of the convex hull of Λ are the vectors of the form $(0,\ldots,0,1,\ldots,1)$.

Consider an edge $e \in E$. We claim that for this family, on each interval defined by two consecutive points of $B(e)$, the function $R(x)$ can be described as an upper envelope of n convex functions. Indeed, for each $1 \leq s < b(e)$, and for each $x \in [\bar{x}_s^e, \bar{x}_{s+1}^e]$, $R(x) = \max\limits_{k=1,\ldots,n} \left(\Big(\sum\limits_{i=n-k+1}^{n} d_{(i)}(x) \Big) - \zeta_k \right)$, where $\zeta_k = \min\limits_{y\in EQ} \sum\limits_{l=n-k+1}^{n} d_{(l)}(y)$. Hence,

$$R(x) = \max_{k=1,\ldots,n} \Big(S_k(x) - \min_{y\in EQ} S_k(y) \Big).$$

Note that for each $i = 1,\ldots,n$, the function $d(v_i,x)$ is linear over the subinterval $[\bar{x}_s^e, \bar{x}_{s+1}^e]$. Therefore, for each $k = 1,\ldots,n$, the function $\sum_{l=n-k+1}^{n} d_{(l)}(x)$ is convex over the subinterval $[\bar{x}_s^e, \bar{x}_{s+1}^e]$. (See [15,19]).

To evaluate $R(x)$ for a given x, it is sufficient to sort the elements $\{w_i d(x, v_i)\}$ in order to compute the terms $\{S_k(x)\}_k$, and finally find $\max_{k=1,\ldots,n} \Big(S_k(x) - \min_{y \in EQ} S_k(y)\Big)$. The minimum of $R(x)$ on the interval $[\bar{x}^e_s, \bar{x}^e_{s+1}]$ can then be computed in $O(n \log^2 n)$ time by using the parametric approach of Megiddo [13] with the modification in Cole [7]. Hence, the solution to the minimax regret ordered median problem on a given edge e for this family of functions can be obtained in $O(n \log^2 n|B(e)|)$ time. (We assume that in the preprocessing phase of the algorithm we have already calculated the terms $\{\zeta_k\}_k$. The total effort for this phase is $O(mn^2 \log n)$, see [15]).

Theorem 3. *The total time to solve the minimax regret k-centrum problem on a general graph is* $O(mn^2 \log^2 n)$.

The above analysis and algorithm are also applicable to the more general convex case defined by $\Lambda = \{(\lambda_1, \ldots, \lambda_n) \,:\, \lambda_1 \leq \ldots \leq \lambda_n \text{ and } \lambda_i \in [a, b],\ i = 1, \ldots, n\}$, where a and b satisfy $0 \leq a \leq b$, see [15]. In this case we have

$$\Lambda = CH(\{(\lambda_1, \ldots, \lambda_n) \,:\, \lambda_1 \leq \ldots \leq \lambda_n \text{ and } \lambda_i \in \{a, b\},\ i = 1, \ldots, n\}).$$

As another example of a convex family of ordered median functions with a small number of extreme points, consider the model corresponding to the α-centdian problem, see [15]. In this case, $\Lambda = \{(\lambda_1, \ldots, \lambda_n) \,:\, \exists\, 0 \leq \alpha \leq 1,\ \lambda_1 = \ldots = \lambda_{n-1} = \alpha,\ \text{ and } \lambda_n = 1\}$. We have $ext(\Lambda) = \{(0, \ldots, 0, 1), (1, \ldots, 1)\}$.

5.1 The Case of Interval Weights and Order Constraints

In this section, we consider the minimax regret convex ordered median problem where the $\lambda-$weights are in the set,

$$\Lambda_{\leq} = \{(\lambda_1, \ldots, \lambda_n) \,:\, \lambda_i \in [a_i, b_i] \ \text{ for } i = 1, \ldots, n, \text{ and } \ 0 \leq \lambda_1 \leq \ldots \leq \lambda_n\}.$$

Without loss of generality, we may assume that both sequences $\{a_i\}$ and $\{b_i\}$ are nonnegative and nondecreasing. We note that the components of each extreme point of $\Lambda_{\leq}$ are elements of the set $AB = \{a_1, \ldots, a_n, b_1, \ldots, b_n\}$. Indeed, let λ be an extreme point and suppose without loss of generality that some $\lambda_i \notin AB$. Let $1 \leq s \leq i \leq t \leq n$ be such that $\lambda_{s-1} < \lambda_s = \lambda_i = \lambda_t < \lambda_{t+1}$. For $\varepsilon > 0$ sufficiently small, consider the vector $\lambda(\varepsilon+)$ defined by setting $\lambda_j(\varepsilon+) = \lambda_j + \varepsilon$ for $s \leq j \leq t$ and $\lambda_j(\varepsilon+) = \lambda_j$ otherwise. Similarly, consider the vector $\lambda(\varepsilon-)$ defined by setting $\lambda_j(\varepsilon-) = \lambda_j - \varepsilon$ for $s \leq j \leq t$ and $\lambda_j(\varepsilon-) = \lambda_j$ otherwise. The vector λ is the midpoint of the interval connecting $\lambda(\varepsilon+)$ and $\lambda(\varepsilon-)$, contradicting the fact that λ is an extreme point of $\Lambda_{\leq}$.

Recall that for a given $y \in EQ$, $R_y(x) = \max_{\lambda \in \Lambda_{\leq}} \Big(f_\lambda(x) - f_\lambda(y)\Big)$, see (11). Evaluating $R(x)$ for a given x amounts to computing the $|EQ|$ values $R_y(x)$ for all $y \in EQ$, see (2).

We propose an algorithm to compute $R_y(x)$ for any fixed $y \in EQ$. Consider an edge $e \in E$. Let $\bar{x}^e_k$ and $\bar{x}^e_{k+1}$ be two consecutive elements of $B(e)$. $f_\lambda(x)$ is a

convex function on $[\bar{x}^e_k, \bar{x}^e_{k+1}]$, see [15]. By Lemma 2, the functions $\{R_y(x)\}_{y\in EQ}$, as well as $R(x)$, are all piecewise linear and convex on $[\bar{x}^e_k, \bar{x}^e_{k+1}]$.

For a fixed $y \in EQ$ and $x \in [\bar{x}^e_k, \bar{x}^e_{k+1}]$ the evaluation of $R_y(x)$ can be done by solving the following linear program:

$$\begin{aligned} R_y(x) = \max\ & c^T(x)\lambda - h^T(y)\lambda \\ s.t.\ & \lambda_i - \lambda_{i+1} \le 0, \quad \forall i = 1,\dots,n-1, \\ & a_i \le \lambda_i \le b_i, \quad \forall i = 1,\dots,n, \end{aligned}$$

where $c^T(x) = (d_{(1)}(x),\dots,d_{(n)}(x))$ and $h^T(y) = (d_{(1)}(y),\dots,d_{(n)}(y))$. (Recall that the optimal solution λ^* satisfies $\lambda^*_i \in AB$ for $i = 1,\dots,n$).

Defining $\mu_i = \lambda_i - a_i$, $\beta_i = b_i - a_i$, for $i = 1,\dots,n$, $\alpha_n = 0$, and $\alpha_i = a_{i+1} - a_i$ for $i = 1,\dots,n-1$, the formulation above reduces to:

$$\begin{aligned} \sum_{i=1}^{n} a_i(d_{(i)}(x) - d_{(i)}(y)) + \max\ & c^T(x)\mu - h^T(y)\mu \\ s.t.\ & \mu_i - \mu_{i+1} \le \alpha_i, \quad \forall i = 1,\dots,n-1, \\ & \mu_i \le \beta_i, \quad \forall i = 1,\dots,n, \\ & \mu_i \ge 0, \quad \forall i = 1,\dots,n. \end{aligned}$$

Setting $u_0 = u_n = 0$, the formulation of its corresponding dual problem is:

$$\begin{aligned} \sum_{i=1}^{n} a_i(d_{(i)}(x) - d_{(i)}(y)) + \min\ & \sum_{i=1}^{n} \alpha_i u_i + \sum_{i=1}^{n} \beta_i t_i \\ s.t.\ & u_i - u_{i-1} + t_i \ge d_{(i)}(x) - d_{(i)}(y), \quad \forall i = 1,\dots,n, \\ & u_i, t_i \ge 0, \quad i = 1,\dots,n. \end{aligned}$$

Notice that the matrix defining the above linear program is totally unimodular since it is a flow matrix augmented by the identity matrix.

The above model corresponds to the following single commodity min-cost flow problem. (See Figure 2.) For each $i = 1,\dots,n-1$, u_i is the flow on the arc $(i+1, i)$, and for each $i = 1,\dots,n$, t_i is the flow on left-arc $(0, i)$. For $i = 1,\dots,n$, the demand at node i is $d_{(i)}(x) - d_{(i)}(y)$. Note that the demand can be of any sign. To account for the sign of the demand for $i = 1,\dots,n$, define $\delta^+_i(x) = \max\{0, d_{(i)}(x) - d_{(i)}(y)\}$ and $\delta^-_i(x) = \max\{0, -(d_{(i)}(x) - d_{(i)}(y))\}$. The label attached to each edge in the graph has two coordinates. The left coordinate is the capacity upper bound on the flow, and the right one is the per unit cost of flow on the edge. The min-cost flow problem is to find the minimum cost of transporting $\sum_{i=1}^{n} \delta^+_i(x)$ units from the source node 0 to the destination $n+1$.

The fastest known strongly polynomial algorithm to solve a single commodity min-cost flow problem on a network with n nodes and m edges is $O(mS(n,m)\log n)$, where $S(n,m)$ is the time to solve the single source shortest path problem on a network with n nodes and m edges, having nonnegative lengths. (See Ahuja et al. [1].) This algorithm has $O(m \log n)$ scaling phases, where in each phase a shortest path problem is solved in $S(n,m)$ time. For a

general graph $S(n,m) = O(m + n\log n)$ time. In our case m, the number of edges, satisfies $m \leq 4n$, and $S(n,m) = O(n\log n)$. Moreover, the above flow problem is defined on a special planar network. It is a 3-tree, i.e., its tree-width is bounded by 3. The latter bound follows from the fact that this network has a separator consisting of the three nodes, $\{0, \lfloor n/2 \rfloor, n+1\}$. (Each of the two connected components obtained by removing these three nodes is a path consisting of at most $n/2$ nodes. In particular, each component has a separator consisting of its median node).

Using the linear time algorithm in Henzinger et al., [10], designed for general planar graphs, or the more special algorithm for graphs with bounded tree-width in Chaudhuri and Zaroliagis, [6], in our case we have $S(n,m) = O(n)$. We conclude that computing $R_y(x)$ for a given x and $y \in EQ$ can be done in $O(n^2 \log n)$ time. Moreover, applying the results in [6], $R_y(x)$ can also be computed in $O(n\log^2 n)$ parallel time with $O(n/\log n)$ processors. With the above tools we can then directly use the parametric approach in Megiddo [13] with the modification in Cole [7], to obtain the next result.

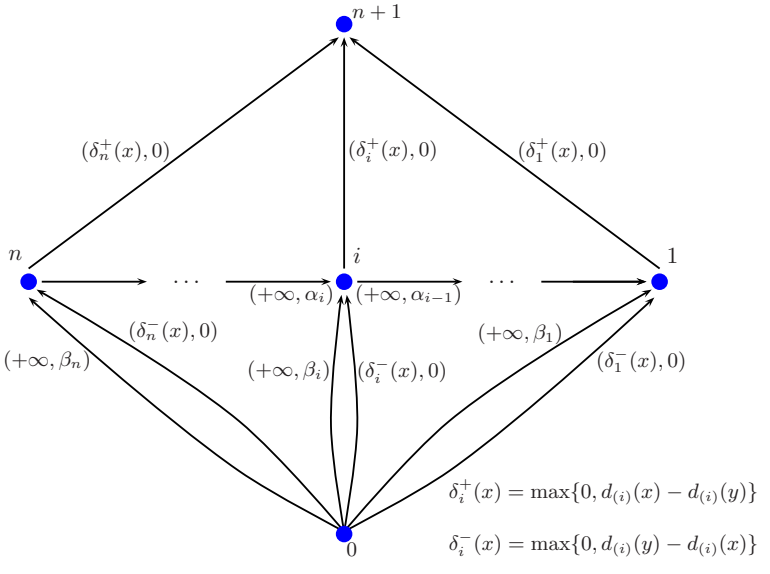


Fig. 2. Illustration of a n-diamond type network

Theorem 4. *The total time to solve the single facility minimax regret convex ordered median problem on a general graph is* $O(m^2 n^6 \log^4 n)$.

Finally, we use global convexity properties of the function $R(x)$ on tree graphs to solve the minimax regret convex ordered median problem on such graphs more efficiently.

Theorem 5. *The total time to solve the single facility minimax regret convex ordered median problem on a tree graph is* $O(n^6 \log^4 n)$. *If the node weights are identical the total time reduces to* $O(n^5 \log^4 n)$.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows. Prentice Hall, New Jersey (1993)
2. Aissi, H., Bazgan, C., Vanderpooten, D.: Approximation of min-max and min-max regret versions of some combinatorial optimization problems. Eur. J. Oper. Res. 179, 281–290 (2007)
3. Averbakh, I.: On the complexity of a class of combinatorial optimization problems with uncertainty. Math. Program. 90, 263–272 (2001)
4. Averbakh, I., Berman, O.: An improved algorithm for the minmax regret median problem on a tree. Networks 41(2), 97–103 (2003)
5. Averbakh, I., Berman, O.: On the complexity of minmax regret linear programming. Eur. J. Oper. Res. 160, 227–231 (2005)
6. Chaudhuri, S., Zaroliagis, C.: Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. Algorithmica. 27, 212–226
7. Cole, R.: Slowing down sorting networks to obtain faster sorting algorithms. J. Assoc. Comput. Mach. 34, 200–208 (1987)
8. Conde, E.: An improved algorithm for selecting p items with uncertain returns according to the minmax-regret criterion. Math. Program 100, 345–353 (2004)
9. Francis, R.L., Lowe, T.J., Tamir, A.: Aggregation error bounds for a class of location models. Oper. Res. 48, 294–307 (2000)
10. Henzinger, M.R., Klein, P., Rao, S., Subramanian, S.: Faster shortest-path algorithms for planar graphs. Journal of Computers and System Sciences 55, 3–23 (1997)
11. Hershberger, J., Suri, S.: Off-line maintenance of planar configurations. J. Algorithms 21(3), 453–475 (1996)
12. Kouvelis, P., Yu, G.: Robust discrete optimization and its applications. Kluwer Academic Publishers, Dordrecht (1997)
13. Megiddo, N.: Applying parallel computation algorithms in the design of serial algorithms. J. Assoc. Comput. Mach. 30, 852–865 (1983)
14. Megiddo, N.: Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. SIAM J. Comput. 12, 759–776 (1983)
15. Nickel, S., Puerto, J.: Location Theory: A unified approach. Springer, Heidelberg (2005)
16. Puerto, J., Rodriguez-Chia, A.M., Tamir, A.: Minimax regret single facility ordered median location problems on networks. Preprint Facultad de Matemáticas, Universidad de Sevilla (2007)
17. Ravi, R., Sinha, A.: Hedging uncertainty: approximation algorithms for stochastic optimization problems. Math. Program. 108, 97–114 (2006)
18. Tamir, A.: On the solution value of the continuous p-center location problem on a graph. Math. Oper. Res. 12, 340–349 (1987)
19. Tamir, A.: The k-centrum multi-facility location problem. Discrete Appl. Math. 109, 292–307 (2000)

Dial a Ride from k-Forest

Anupam Gupta*, MohammadTaghi Hajiaghayi**,
Viswanath Nagarajan***, and R. Ravi***

Carnegie Mellon University, Pittsburgh PA 15213, USA

Abstract. The *k-forest problem* is a common generalization of both the *k-MST* and the *dense-k-subgraph* problems. Formally, given a metric space on n vertices V, with m demand pairs $\subseteq V \times V$ and a "target" $k \leq m$, the goal is to find a minimum cost subgraph that connects *at least* k demand pairs. In this paper, we give an $O(\min\{\sqrt{n}, \sqrt{k}\})$-approximation algorithm for k-forest, improving on the previous best ratio of $O(\min\{n^{2/3}, \sqrt{m}\} \log n)$ by Segev and Segev [20].

We then apply our algorithm for k-forest to obtain approximation algorithms for several *Dial-a-Ride* problems. The basic Dial-a-Ride problem is the following: given an n point metric space with m objects each with its own source and destination, and a vehicle capable of carrying *at most* k objects at any time, find the minimum length tour that uses this vehicle to move each object from its source to destination. We prove that an α-approximation algorithm for the k-forest problem implies an $O(\alpha \cdot \log^2 n)$-approximation algorithm for Dial-a-Ride. Using our results for k-forest, we get an $O(\min\{\sqrt{n}, \sqrt{k}\} \cdot \log^2 n)$-approximation algorithm for Dial-a-Ride. The only previous result known for Dial-a-Ride was an $O(\sqrt{k} \log n)$-approximation by Charikar and Raghavachari [5]; our results give a different proof of a similar approximation guarantee—in fact, when the vehicle capacity k is large, we give a slight improvement on their results.

The reduction from Dial-a-Ride to the k-forest problem is fairly robust, and allows us to obtain approximation algorithms (with the same guarantee) for the following generalizations: (i) Non-uniform Dial-a-Ride, where the cost of traversing each edge is an arbitrary non-decreasing function of the number of objects in the vehicle; and (ii) Weighted Dial-a-Ride, where demands are allowed to have different weights. The reduction is essential, as it is unclear how to extend the techniques of Charikar and Raghavachari to these Dial-a-Ride generalizations.

1 Introduction

In the Steiner forest problem, we are given a set of vertex-pairs, and the goal is to find a forest such that each vertex pair lies in the same tree in the forest. This

* Supported in part by an NSF CAREER award CCF-0448095, and by an Alfred P. Sloan Fellowship.
** Supported in part by NSF ITR grant CCR-0122581 (The ALADDIN project).
*** Supported in part by NSF grants CCF-0430751 and ITR grant CCR-0122581 (The ALADDIN project).

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 241–252, 2007.

is a generalization of the Steiner tree problem, where all the pairs contain a common vertex called the root; both the tree and forest versions are well-understood fundamental problems in network design, and constant factor approximation algorithms are known [17,1,12]. An important extension of the Steiner tree problem studied in the late 1990s was the k-MST problem, where one sought the least-cost tree that connected any k of the terminals: several approximations algorithms were given for the problem, culminating in the 2-approximation of Garg [11]; the k-MST problem proved crucial in many subsequent developments in network design and vehicle routing [6,8,3,2]. One can analogously define the k-forest problem where one needs to connect *only k of the pairs* in some Steiner forest instance: surprisingly, very little is known about this problem, which was first studied formally as recently as last year [15,20]. In this paper, we give a simpler and improved approximation algorithm for the k-forest problem.

Moreover, just like the k-MST variant, the k-forest problem seems to be useful in applications to network design and vehicle routing. In the second half of the paper, we show a (somewhat surprising) reduction of a well-studied vehicle routing problem called the Dial-a-Ride problem to the k-forest problem. In the Dial-a-Ride problem, we are given a metric space with people having sources and destinations, and a bus of some capacity k; the goal is to find a route for this bus so that each person can be taken from her source to destination without exceeding the capacity of the bus at any point, such that the length of the bus route is minimized. We show how the results for the k-forest problem slightly improve upon existing results for the Dial-a-Ride problem; in fact, they give the first approximation algorithms for some generalizations of Dial-a-Ride which do not seem amenable to previous techniques.

1.1 The k-Forest Problem

Our starting point is the k-forest problem, which generalizes both the k-MST and the dense-k-subgraph problems.

Definition 1 (The k-Forest Problem). *Given an n-vertex metric space (V, d), and* demands $\{s_i, t_i\}_{i=1}^{m} \subseteq V \times V$*, find the least-cost subgraph that connects at least k demand-pairs.*

Note that the k-forest problem is a generalization of the (minimization version of the) well-studied dense-k-subgraph problem, for which nothing better than an $O(n^{1/3-\delta})$ approximation is known ($\delta > 0$ is some constant). The k-forest problem was first defined in [15], and the first non-trivial approximation was given by Segev and Segev [20], who gave an algorithm with an approximation guarantee of $O(n^{2/3} \log n)$. We improve the approximation guarantee of the k-forest problem to $O(\min\{\sqrt{n}, \sqrt{k}\})$; formally, we prove the following theorem in Section 2.

Theorem 2 (Approximating k-forest). *There is an $O(\min\{\sqrt{n} \cdot \frac{\log k}{\log n}, \sqrt{k}\})$-approximation algorithm for the k-forest problem. For the case when k is less than a polynomial in n, the approximation guarantee improves to $O(\min\{\sqrt{n}, \sqrt{k}\})$.*

Apart from giving an improved approximation guarantee, our algorithm for the k-forest problem is arguably simpler and more direct than that of [20] (which is based on Lagrangian relaxations for the problem, and combining solutions to this relaxation). Indeed, we give two algorithms, both reducing the k-forest problem to the k-MST problem in different ways and achieving different approximation guarantees—we then return the better of the two answers. The first algorithm (giving an approximation of $O(\sqrt{k})$) uses the k-MST algorithm to find good solutions on the sources and the sinks independently, and then uses the Erdős-Szekeres theorem on monotone subsequences to find a "good" subset of these sources and sinks to connect cheaply; details are given in Section 2.1. The second algorithm starts off with a single vertex as the initial solution, and uses the k-MST algorithm to repeatedly find a low-cost tree that satisfies a large number of demands which have one endpoint in the current solution and the other endpoint outside; this tree is then used to greedily augment the current solution and proceed. Choosing the parameters (as described in Section 2.2) gives us an $O(\sqrt{n})$ approximation.

1.2 The Dial-a-Ride Problem

In this paper, we use the k-forest problem to give approximation algorithms for the following vehicle routing problem.

Definition 3 (The Dial-a-Ride Problem). *Given an n-vertex metric space (V, d), a starting vertex (or* root*) r, a set of m demands $\{(s_i, t_i)\}_{i=1}^{m}$, and a vehicle of capacity k, find a minimum length tour of the vehicle starting (and ending) at r that moves each object i from its source s_i to its destination t_i such that the vehicle carries at most k objects at any point on the tour.*

We say that an object is *preempted* if, after being picked up from its source, it can be left at some intermediate vertices before being delivered to its destination. In this paper, we will not allow this, and will mainly be concerned with the *non-preemptive* Dial-a-Ride problem.[1]

The approximability of the Dial-a-Ride problem is not very well understood: the previous best upper bound is an $O(\sqrt{k}\log n)$-approximation algorithm due to Charikar and Raghavachari [5], whereas the best lower bound that we are aware of is APX-hardness (from TSP, say). We establish the following (somewhat surprising) connection between the Dial-a-Ride and k-forest problems in Section 3.

Theorem 4 (Reducing Dial-a-Ride to k-forest). *Given an α-approximation algorithm for k-forest, there is an $O(\alpha \cdot \log^2 n)$-approximation algorithm for the Dial-a-Ride problem.*

[1] A note on the parameters: a feasible non-preemptive tour can be short-cut over vertices that do not participate in any demand, and we can assume that every vertex is an end point of some demand; so $n \leq 2m$. We may also assume, by preprocessing some demands, that $m \leq n^2 \cdot k$. However in general, the number of demands m and the vehicle capacity k may be much larger than the number of vertices n.

In particular, combining Theorems 2 and 4 gives us an $O(\min\{\sqrt{k}, \sqrt{n}\} \cdot \log^2 n)$-approximation guarantee for Dial-a-Ride. Of course, improving the approximation guarantee for k-forest would improve the result for Dial-a-Ride as well.

Note that our results match the results of [5] up to a logarithmic term, and even give a slight improvement when the vehicle capacity $k \gg n$, the number of nodes. Much more interestingly, our algorithm for Dial-a-Ride easily extends to generalizations of the Dial-a-Ride problem. In particular, we consider a substantially more general vehicle routing problem where the vehicle has no *a priori* capacity, and instead the cost of traversing each edge e is an arbitrary non-decreasing function $c_e(l)$ of the number of objects l in the vehicle; setting $c_e(l)$ to the edge-length d_e when $l \leq k$, and $c_e(l) = \infty$ for $l > k$ gives us back the classical Dial-a-Ride setting. We show that this general *non-uniform Dial-a-Ride* problem admits an approximation guarantee that matches the best known for the classical Dial-a-Ride problem. Another extension we consider is the *weighted Dial-a-Ride* problem. In this, each object may have a different size, and total size of the items in the vehicle must be bounded by the vehicle capacity; this has been earlier studied as the *pickup and delivery* problem [19]. We show that this problem can be reduced to the (unweighted) Dial-a-Ride problem at the loss of only a constant factor in the approximation guarantee.

As an aside, we consider the effect of the number of preemptions in the Dial-a-Ride problem. It was shown in Charikar and Raghavachari [5] that the gap between the optimal preemptive and non-preemptive tours could be as large as $\Omega(n^{1/3})$. We show that the real difference arises between *zero* and *one* preemptions: allowing multiple preemptions does not give us much added power. In particular, we show that for any instance of the Dial-a-Ride problem, there is a tour that preempts each object *at most once* and has length at most $O(\log^2 n)$ times an optimal preemptive tour (which may preempt each object an arbitrary number of times).

Due to lack of space, the results on extensions of Dial-a-Ride and the effect of preemptions are deferred to the full version of the paper [13].

1.3 Related Work

The k-forest problem: The k-forest problem is relatively new: it was defined by Hajiaghayi and Jain [15]. An $\tilde{O}(k^{2/3})$-approximation algorithm for even the directed k-forest problem can be inferred from [4]. Recently, Segev and Segev [20] gave an $O(\min\{n^{2/3}, \sqrt{m}\} \log n)$ approximation algorithm for k-forest.

Dense k-subgraph: The *k-forest* problem is a generalization of the dense-k-subgraph problem [9], as shown in [15]. The best known approximation guarantee for the dense-k-subgraph problem is $O(n^{1/3-\delta})$ where $\delta > 0$ is some constant, due to Feige et al. [9], and obtaining an improved guarantee has been a long standing open problem. Strictly speaking, Feige et al. [9] study a potentially harder problem: the *maximization* version of dense-k-subgraph, where one wants to pick k vertices to maximize the number of edges in the induced graph. However, nothing better is known even for the *minimization* version of dense-k-subgraph

(where one wants to pick the minimum number of vertices that induce k edges), which is a special case of k-forest. The k-forest problem is also a generalization of k-MST, for which a 2-approximation is known (Garg [11]).

Dial-a-Ride: While the Dial-a-Ride problem has been studied extensively in the operations research literature, relatively little is known about its approximability. The currently best known approximation ratio for Dial-a-Ride is $O(\sqrt{k}\log n)$ due to Charikar and Raghavachari [5]. We note that their algorithm assumes instances with unweighted demands. Krumke et al. [16] give a 3-approximation algorithm for the Dial-a-Ride problem on a *line metric*; in fact, their algorithm finds a non-preemptive tour that has length at most 3 times the preemptive lower bound. (Clearly, the cost of an optimal preemptive tour is at most that of an optimal non-preemptive tour.) A 2.5-approximation algorithm for *single source* version of Dial-a-Ride (also called the "capacitated vehicle routing" problem) was given by Haimovich and Kan [14]; again, their algorithm outputs a non-preemptive tour with length at most 2.5 times the preemptive lower bound. The $k = 1$ special case of Dial-a-Ride is also known as the *stacker-crane* problem, for which a 1.8-approximation is known [10]. For the *preemptive* Dial-a-Ride problem, [5] gave the current-best $O(\log n)$ approximation algorithm, and Gørtz [18] showed that it is hard to approximate this problem to better than $\Omega(\log^{1/4-\epsilon} n)$, for any constant $\epsilon > 0$. Recall that no super-constant hardness results are known for the non-preemptive Dial-a-Ride problem.

2 The k-Forest Problem

In this section, we study the k-forest problem, and give an approximation guarantee of $O(\min\{\sqrt{n}, \sqrt{k}\})$. This result improves upon the previous best $O(n^{2/3}\log n)$-approximation guarantee [20] for this problem. The algorithm in Segev and Segev [20] is based on a Lagrangian relaxation for this problem, and suitably combining solutions to this relaxation. In contrast, our algorithm uses a more direct approach and is much simpler in description. Our approach is based on approximating the following "density" variant of k-forest.

Definition 5 (Minimum-ratio k-forest). *Given an n-vertex metric space (V, d), m pairs of vertices $\{s_i, t_i\}_{i=1}^m$, and a target k, find a tree T that connects* at most *k pairs, and minimizes the ratio of the length of T to the number of pairs connected in T.*[2]

We present two different algorithms for *minimum-ratio k-forest*, obtaining approximation guarantees of $O(\sqrt{k})$ (Section 2.1) and $O(\sqrt{n})$ (Section 2.2); these are then combined to give the claimed result for the k-forest problem. Both our algorithms are based on subtle reductions to the k-MST problem, albeit in very different ways.

As is usual, when we say that our algorithm *guesses* a parameter in the following discussion, it means that the algorithm is run for each possible value of

[2] Even if we relax the solution to be any forest, we may assume (by averaging) that the *optimal ratio* solution is a tree.

that parameter, and the best solution found over all the runs is returned. As long as only a constant number of parameters are being guessed and the number of possibilities for each of these parameters is polynomial, the algorithm is repeated only a polynomial number of times.

2.1 An $O(\sqrt{k})$ Approximation Algorithm

In this section, we give an $O(\sqrt{k})$ approximation algorithm for minimum ratio k-forest, which is based on a simple reduction to the k-MST problem. The basic intuition is to look at the solution S to minimum-ratio k-forest and consider an Euler tour of this tree S—a theorem of Erdős and Szekeres on increasing subsequences implies that there must be at least $\sqrt{|S|}$ sources which are visited in the same order as the corresponding sinks. We use this existence result to combine the source-sink pairs to create an instance of $\sqrt{|S|}$-MST from which we can obtain a good-ratio tree; the details follow.

Let S denote an optimal ratio tree, that covers q demands and has length B, and let D denote the largest distance between any demand pair that is covered in S (note $D \leq B$). We define a new metric l on the set $\{1, \cdots, m\}$ of demands as follows. The distance between demands i and j, $l_{i,j} = d(s_i, s_j) + d(t_i, t_j)$, where (V, d) is the original metric. The $O(\sqrt{k})$ approximation algorithm first guesses the number of demands q and the largest demand-pair distance D in the optimal tree S (there are at most m choices for each of q & D). The algorithm discards all demand pairs (s_i, t_i) such that $d(s_i, t_i) > D$ (all the pairs covered in the optimal solution S still remain). Then the algorithm runs the unrooted k-MST algorithm [11] with target $\lfloor\sqrt{q}\rfloor$, in the metric l, to obtain a tree T on the demand pairs P. From T, we easily obtain trees T_1 (on all sources in P) and T_2 (on all sinks in P) in metric d such that $d(T_1) + d(T_2) = l(T)$. Finally the algorithm outputs the tree $T' = T_1 \cup T_2 \cup \{e\}$, where e is any edge joining a source in T_1 to its corresponding sink in T_2. Due to the pruning on demand pairs that have large distance, $d(e) \leq D$ and the length of T', $d(T') \leq l(T) + D \leq l(T) + B$.

We now argue that the cost of the solution T found by the k-MST algorithm $l(T) \leq 8B$. Consider the optimal ratio tree S (in metric d) that has q demands $\{(s_1, t_1), \cdots, (s_q, t_q)\}$, and let τ denote an Euler tour of S. Suppose that in a traversal of τ, the *sources* of demands in S are seen in the order $s_1, \cdots, s_q$. Then in the same traversal, the *sinks* of demands in S will be seen in the order $t_{\pi(1)}, \cdots, t_{\pi(q)}$, for some permutation π. The following fact is well known (see, e.g., [21]).

Theorem 6 (Erdős and Szekeres). *Every permutation on* $\{1, \cdots, q\}$ *has either an increasing or a decreasing subsequence of length* $\lfloor\sqrt{q}\rfloor$.

Using Theorem 6, we obtain a set M of $p = \lfloor\sqrt{q}\rfloor$ demands such that (1) the sources in M appear in increasing order in a traversal of the Euler tour τ, and (2) the sinks in M appear in increasing order in a traversal of either τ or τ^R (the reverse traversal of τ). Let $j_0 < j_1 < \cdots < j_{p-1}$ denote the demands in M in increasing order. From statement (1) above, $\sum_{i=0}^{p-1} d(s(j_i), s(j_{i+1})) \leq d(\tau)$, where

the indices in the summation are modulo p. Similarly, statement (2) implies that $\sum_{i=0}^{p-1} d(t(j_i), t(j_{i+1})) \leq \max\{d(\tau), d(\tau^R)\} = d(\tau)$. Thus we obtain:

$$\sum_{i=0}^{p-1} [d(s(j_i), s(j_{i+1})) + d(t(j_i), t(j_{i+1}))] \leq 2d(\tau) \leq 4B$$

But this sum is precisely the length of the tour $j_0, j_1, \cdots, j_{p-1}, j_0$ in metric l. In other words, there is a tree of length $4B$ in metric l, that contains $\lfloor \sqrt{q} \rfloor$ vertices. So, the cost of the solution T found by the k-MST approximation algorithm is at most $8B$.

Now the final solution T' has length at most $l(T) + B \leq 9B$, and ratio that at most $9\sqrt{q}\frac{B}{q} \leq 9\sqrt{k}\frac{B}{q}$. Thus we have an $O(\sqrt{k})$ approximation algorithm for minimum ratio k-forest.

2.2 An $O(\sqrt{n})$ Approximation Algorithm

In this section, we show an $O(\sqrt{n})$ approximation algorithm for the minimum ratio k-forest problem. The approach is again to reduce to the k-MST problem; the intuition is rather different: either we find a vertex v such that a large number of demand-pairs of the form $(v, *)$ can be satisfied using a small tree (the "high-degree" case); if no such vertex exists, we show that a repeated greedy procedure would cover most vertices without paying too much (and since we are in the "low-degree" case, covering most vertices implies covering most demands too). The details follow.

Let S denote an optimal solution to minimum ratio k-forest, and $q \leq k$ the number of demand pairs covered in S. We define the *degree* Δ of S to be the maximum number of demands (among those covered in S) that are incident at any vertex in S. The algorithm first guesses the following parameters of the optimal solution S: its length B (within a factor 2), the number of pairs covered q, the degree Δ, and the vertex $w \in S$ that has Δ demands incident at it. Although, there may be an exponential number of choices for the optimal length, a polynomial number of guesses within a binary-search suffice to get a B such that $B \leq d(S) \leq 2 \cdot B$. The algorithm then returns the better of the two procedures described below.

Procedure 1 (high-degree case): Since the degree of vertex w in the optimal solution S is Δ, there is tree rooted at w of length $d(S) \leq 2B$, that contains at least Δ demands having one end point at w. We assign a weight to each vertex u, equal to the number of demands that have one end point at this vertex u and the other end point at w. Then we run the k-MST algorithm [11] with root w and a target weight of Δ. By the preceding argument, this problem has a feasible solution of length $2B$; so we obtain a solution H of length at most $4B$ (since the algorithm of [11] is a 2-approximation). The ratio of solution H is thus at most $4B/\Delta = \frac{4q}{\Delta}\frac{B}{q}$.

Procedure 2 (low-degree case): Set $t = \frac{q}{2\Delta}$; note that $q \leq \frac{\Delta \cdot n}{2}$ and so $t \leq n/4$. We maintain a current tree T (initially just vertex w), which is updated in iterations as follows: shrink T to a supernode s, and run the k-MST algorithm with root s and a target of t new vertices. If the resulting s-tree has length at most $4B$, include this tree in the current tree T and continue. If the resulting s-tree has length more than $4B$, or if all the vertices have been included, the procedure ends. Since t new vertices are added in each iteration, the number of iterations is at most $\frac{n}{t}$; so the length of T is at most $\frac{4n}{t}B$. We now show that T contains at least $\frac{q}{2}$ demands. Consider the set $S \setminus T$ (recall, S is the optimal solution). It is clear that $|V(S) \setminus V(T)| < t$; otherwise the k-MST instance in the last iteration (with the current T) would have S as a feasible solution of length at most $2B$ (and hence would find one of length at most $4B$). So the number of demands covered in S that have at least one end point in $S \setminus T$ is at most $|V(S) \setminus V(T)| \cdot \Delta \leq t \cdot \Delta = q/2$ (as Δ is the degree of solution S). Thus there are at least $q/2$ demands *contained* in $S \cap T$, in particular in T. Thus T is a solution having ratio at most $\frac{4n}{t}B \cdot \frac{2}{q} = \frac{8n}{t}\frac{B}{q}$.

The better ratio solution among H and T from the two procedures has ratio at most $\min\{\frac{4q}{\Delta}, \frac{8n}{t}\} \cdot \frac{B}{q} = \min\{8t, \frac{8n}{t}\} \cdot \frac{B}{q} \leq 8\sqrt{n} \cdot \frac{B}{q} \leq 8\sqrt{n} \cdot \frac{d(S)}{q}$. So this algorithm is an $O(\sqrt{n})$ approximation to the minimum ratio k-forest problem.

2.3 Approximation Algorithm for k-Forest

Given the two algorithms for minimum ratio k-forest, we can use them in a standard greedy fashion (i.e., keep picking approximately minimum-ratio solutions until we obtain a forest connecting at least k pairs); the standard set cover analysis can be used to show an $O(\min\{\sqrt{n}, \sqrt{k}\} \cdot \log k)$-approximation guarantee for k-forest. A tighter analysis of the greedy algorithm (as done, e.g., in Charikar et al. [4]) can be used to remove the logarithmic terms and obtain the guarantee stated in Theorem 2.

3 Application to Dial-a-Ride Problems

In this section, we study an application of the k-forest problem to the Dial-a-Ride problem (Definition 3). A natural solution-structure for Dial-a-Ride involves servicing demands in batches of at most k each, where a batch consisting of a set S of demands is served as follows: the vehicle starts out being empty, picks up each of the $|S| \leq k$ objects from their sources, then drops off each object at its destination, and is again empty at the end. If we knew that the optimal solution has this structure, we could obtain a greedy framework for Dial-a-Ride by repeatedly finding the best 'batch' of k demands. However, the optimal solution may involve carrying almost k objects at every point in the tour, in which case it can not be decomposed to be of the above structure. In Theorem 7, we show that there is always a near optimal solution having this 'pick-drop in batches' structure. Using Theorem 7, we then obtain an approximation algorithm for the Dial-a-Ride problem.

Theorem 7 (Structure Theorem). *Given any instance of Dial-a-Ride, there exists a feasible tour τ satisfying the following conditions:*

1. *τ can be split into a set of segments $\{S_1, \cdots, S_t\}$ (i.e., $\tau = S_1 \cdot S_2 \cdots S_t$) where each segment S_i services a set O_i of at most k demands such that S_i is a path that first picks up each demand in O_i and then drops each of them.*
2. *The length of τ is at most $O(\log m)$ times the length of an optimal tour.*

Proof: Consider an optimal non-preemptive tour σ: let $c(\sigma)$ denote its length, and $|\sigma|$ denote the number of edge traversals in σ. Note that if in some visit to a vertex v in σ there is no pick-up or drop-off, then the tour can be short-cut over vertex v, and it still remains feasible. Further, due to triangle inequality, the length $c(\sigma)$ does not increase by this operation. So we may assume that each vertex visit in σ involves a pick-up or drop-off of some object. Since there is exactly one pick-up & drop-off for each object, we have $|\sigma| \leq 2m + 1$. Define the *stretch* of a demand i to be the number of edge traversals in σ between the pick-up and drop-off of object i. The demands are partitioned as follows: for each $j = 1, \cdots, \lceil \log(2m) \rceil$, group G_j consists of all the demands whose stretch lie in the interval $[2^{j-1}, 2^j)$. We consider each group G_j separately.

Claim 8. *For each $j = 1, \cdots, \lceil \log(2m) \rceil$, there is a tour τ_j that serves all the demands in group G_j, satisfies condition 1 of Theorem 7, and has length at most $6 \cdot c(\sigma)$.*

Proof: Consider tour σ as a line $\mathcal{L}$, with every edge traversal in σ represented by a distinct edge in $\mathcal{L}$. Number the vertices in $\mathcal{L}$ from 0 to h, where $h = |\sigma|$ is the number of edge traversals in σ. Note that each vertex in V may be represented multiple times in $\mathcal{L}$. Each demand is associated with the numbers of the vertices (in $\mathcal{L}$) where it is picked up and dropped off.

Let $r = 2^{j-1}$, and partition G_j as follows: for $l = 1, \cdots, \lceil \frac{h}{r} \rceil$, set $O_{l,j}$ consists of all demands in G_j that are picked up at a vertex numbered between $(l-1)r$ and $lr - 1$. Since every demand in G_j has stretch in the interval $[r, 2r]$, every demand in $O_{l,j}$ is dropped off at a vertex numbered between lr and $(l+2)r - 1$. Note that $|O_{l,j}|$ equals the number of demands in G_j carried over edge $(lr-1, lr)$ by tour σ, which is at most k. We define segment $S_{l,j}$ to start at vertex number $(l-1)r$ and traverse all edges in $\mathcal{L}$ until vertex number $(l+2)r - 1$ (servicing all demands in $O_{l,j}$ by first picking up each demand between vertices $(l-1)r$ & $lr-1$; then dropping off each demand between vertices lr & $(l+2)r-1$), and then return (with the vehicle being empty) to vertex lr. Clearly, the number of objects carried over any edge in $S_{l,j}$ is at most the number carried over the corresponding edge traversal in σ. Also, each edge in $\mathcal{L}$ participates in at most 3 segments $S_{l,j}$, and each edge is traversed at most twice in any segment. So the total length of all segments $S_{l,j}$ is at most $6 \cdot c(\sigma)$. We define tour τ_j to be the concatenation $S_{1,j} \cdots S_{\lceil h/r \rceil, j}$. It is clear that this tour satisfies condition 1 of Theorem 7. ■

Applying this claim to each group G_j, and concatenating the resulting tours, we obtain the tour τ satisfying condition 1 and having length at most $6 \log(2m) \cdot c(\sigma) = O(\log m) \cdot c(\sigma)$. ■

Remark: The ratio $O(\log m)$ in Theorem 7 is almost best possible. There are instances of Dial-a-Ride (even on an unweighted line), where every solution satisfying condition 1 of Theorem 7 has length at least $\Omega(\max\{\frac{\log m}{\log\log m}, \frac{k}{\log k}\})$ times the optimal non-preemptive tour. So, if we only use solutions of this structure, then it is not possible to obtain an approximation factor (just in terms of capacity k) for Dial-a-Ride that is better than $\Omega(k/\log k)$. The solutions found by the algorithm for Dial-a-Ride in [5] also satisfy condition 1 of Theorem 7. It is interesting to note that when the underlying metric is a hierarchically well-separated tree, [5] obtain a solution of such structure having length $O(\sqrt{k})$ times the optimum, whereas there is a lower bound of $\Omega(\frac{k}{\log k})$ even for the simple case of an unweighted line.

Theorem 7 suggests a greedy strategy for Dial-a-Ride, based on repeatedly finding the best batch of k demands to service. This greedy subproblem turns out to be the minimum ratio k-forest problem (Definition 5), for which we already have an approximation algorithm. The next theorem sets up the reduction from Dial-a-Ride to k-forest.

Theorem 9 (Reducing Dial-a-Ride to minimum ratio k-forest). *An approximation algorithm for minimum ratio k-forest with guarantee ρ implies an $O(\rho\log^2 m)$ approximation algorithm for Dial-a-Ride.*

Proof: The algorithm for Dial-a-Ride is as follows.

1. $\mathcal{C} = \phi$.
2. Until there are no uncovered demands, do:
   (a) Solve the minimum ratio k-forest problem, to obtain a tree C covering $k_C \leq k$ new demands.
   (b) Set $\mathcal{C} \leftarrow \mathcal{C} \cup C$.
3. For each tree $C \in \mathcal{C}$, obtain an Euler tour on C to locally service all demands (pick up all k_C objects in the first traversal, and drop them all in the second traversal). Then use a 1.5-approximate TSP tour on the sources, to connect all the local tours, and obtain a feasible non-preemptive tour.

Consider the tour τ and its segments as in Theorem 7. If the number of uncovered demands in some iteration is m', one of the segments in τ is a solution to the minimum ratio k-forest problem of value at most $\frac{d(\tau)}{m'}$. Since we have a ρ-approximation algorithm for this problem, we would find a segment of ratio at most $O(\rho)\cdot\frac{d(\tau)}{m'}$. Now a standard set cover type argument shows that the total length of trees in $\mathcal{C}$ is at most $O(\rho\log m)\cdot d(\tau) \leq O(\rho\log^2 m)\cdot OPT$, where OPT is the optimal value of the Dial-a-Ride instance. Further, the TSP tour on all sources is a lower bound on OPT, and we use a 1.5-approximate solution [7]. So the final non-preemptive tour output in step 5 above has length at most $O(\rho\log^2 m)\cdot OPT$. ■

This theorem is in fact stronger than Theorem 4 claimed earlier: it is easy to see that any approximation algorithm for k-forest implies an algorithm with the same guarantee for minimum ratio k-forest. Note that, m and k may be

super-polynomial in n. However, it can be shown (see [13]) that with the loss of a constant factor, the general Dial-a-Ride problem can be reduced to one where the number of demands $m \leq n^4$. Based on this and Theorem 9, a ρ approximation algorithm for minimum ratio k-forest actually implies an $O(\rho \log^2 n)$ approximation algorithm for Dial-a-Ride. Using the approximation algorithm for minimum ratio k-forest (Section 2), we obtain an $O(\min\{\sqrt{n}, \sqrt{k}\} \cdot \log^2 n)$ approximation algorithm for the Dial-a-Ride problem.

Remark: If we use the $O(\sqrt{k})$ approximation for k-forest, the resulting non-preemptive tour is in fact feasible even for a $\sqrt{k}$ capacity vehicle! As noted in [5], this property is also true of their algorithm, which is based on an entirely different approach.

References

1. Agrawal, A., Klein, P., Ravi, R.: When trees collide: an approximation algorithm for the generalized steiner problem on networks. In: Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, pp. 134–144. ACM Press, New York (1991)
2. Bansal, N., Blum, A., Chawla, S., Meyerson, A.: Approximation Algorithms for Deadline-TSP and Vehicle Routing with Time Windows. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, pp. 166–174. ACM Press, New York (2004)
3. Blum, A., Chawla, S., Karger, D.R., Lane, T., Meyerson, A., Minkoff, M.: Approximation Algorithms for Orienteering and Discounted-Reward TSP. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 46–55. IEEE Computer Society Press, Los Alamitos (2003)
4. Charikar, M., Chekuri, C., yat Cheung, T., Dai, Z., Goel, A., Guha, S., Li, M.: Approximation algorithms for directed Steiner problems. In: SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, pp. 192–200. ACM Press, New York (1998)
5. Charikar, M., Raghavachari, B.: The Finite Capacity Dial-A-Ride Problem. In: IEEE Symposium on Foundations of Computer Science, pp. 458–467. IEEE Computer Society Press, Los Alamitos (1998)
6. Chaudhuri, K., Godfrey, B., Rao, S., Talwar, K.: Paths, trees, and minimum latency tours. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 36–45 (2003)
7. Christofides, N.: Worst-case analysis of a new heuristic for the travelling salesman problem. GSIA, CMU-Report 388 (1977)
8. Fakcharoenphol, J., Harrelson, C., Rao, S.: The k-traveling repairman problem. In: Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms, pp. 655–664. ACM Press, New York (2003)
9. Feige, U., Peleg, D., Kortsarz, G.: The Dense k -Subgraph Problem. Algorithmica 29(3), 410–421 (2001)
10. Frederickson, G.N., Hecht, M.S., Kim, C.E.: Approximation algorithms for some routing problems. SIAM Journal on Computing 7(2), 178–193 (1978)
11. Garg, N.: Saving an epsilon: a 2-approximation for the k-MST problem in graphs. In: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing, pp. 396–402. ACM Press, New York (2005)

12. Goemans, M.X., Williamson, D.P.: A general approximation technique for constrained forest problems. In: Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 307–316. ACM Press, New York (1992)
13. Gupta, A., Hajiaghayi, M.T., Nagarajan, V., Ravi, R.: Dial a Ride from k-forest (2007), http://arxiv.org/abs/0707.0648
14. Haimovich, M., Rinnooy Kan, A.H.G.: Bounds and heuristics for capacitated routing problems. Mathematics of Operations Research 10, 527–542 (1985)
15. Hajiaghayi, M.T., Jain, K.: The prize-collecting generalized steiner tree problem via a new approach of primal-dual schema. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 631–640. ACM Press, New York (2006)
16. Krumke, S., Rambau, J., Weider, S.: An approximation algorithm for the nonpreemptive capacitated dial-a-ride problem. Preprint 00-53, Konrad-Zuse-Zentrum fr Informationstechnik Berlin (2000)
17. Robins, G., Zelikovsky, A.: Tighter Bounds for Graph Steiner Tree Approximation. SIAM Journal on Discrete Mathematics 19, 122–134 (2005)
18. Gørtz, I.L.: Hardness of Preemptive Finite Capacity Dial-a-Ride. In: 9th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (2006)
19. Savelsbergh, M.W.P., Sol, M.: The general pickup and delivery problem. Transportation Science 29, 17–29 (1995)
20. Segev, D., Segev, G.: Approximate k-Steiner Forests via the Lagrangian Relaxation Technique with Internal Preprocessing. In: 14th Annual European Symposium on Algorithms, pp. 600–611 (2006)
21. Steele, J.M.: Variations on the monotone subsequence theme of Erdős and Szekeres. In: Discrete probability and algorithms (Minneapolis, MN, 1993). IMA Math. Appl., vol. 72, pp. 111–131. Springer, New York (1995)

Online Primal-Dual Algorithms for Maximizing Ad-Auctions Revenue

Niv Buchbinder[1], Kamal Jain[2], and Joseph (Seffi) Naor[1,⋆]

[1] Computer Science Department, Technion, Haifa, Israel
[2] Microsoft Research, Redmond, WA

Abstract. We study the online ad-auctions problem introduced by Mehta et al. [15]. We design a $(1 - 1/e)$-competitive (optimal) algorithm for the problem, which is based on a clean primal-dual approach, matching the competitive factor obtained in [15]. Our basic algorithm along with its analysis are very simple. Our results are based on a unified approach developed earlier for the design of online algorithms [7,8]. In particular, the analysis uses weak duality rather than a tailor made (i.e., problem specific) potential function. We show that this approach is useful for analyzing other classical online algorithms such as ski rental and the TCP-acknowledgement problem. We are confident that the primal-dual method will prove useful in other online scenarios as well.

The primal-dual approach enables us to extend our basic ad-auctions algorithm in a straight forward manner to scenarios in which additional information is available, yielding improved worst case competitive factors. In particular, a scenario in which additional stochastic information is available to the algorithm, a scenario in which the number of interested buyers in each product is bounded by some small number d, and a general risk management framework.

1 Introduction

Maximizing the revenue of a seller in an auction has received much attention recently, and studied in many models and settings. In particular, the way search engine companies such as MSN, Google and Yahoo! maximize their revenue out of selling ad-auctions was recently studied by Mehta *et al.* [15]. In the search engine environment, advertisers link their ads to (search) keywords and provide a bid on the amount paid each time a user clicks on their ad. When users send queries to search engines, along with the (algorithmic) search results returned for each query, the search engine displays funded ads corresponding to *ad-auctions*. The ads are instantly sold, or allocated, to interested advertisers (*buyers*). The total revenue out of this fast growing market is currently billions of dollars. Thus, algorithmic ideas that can improve the allocation of the ads, even by a small percentage, are crucial. The interested reader is refered to [16] for a popular exposition of the ad-auctions problem and the work of [15].

Mehta *et al.* [15] modeled the optimal allocation of ad-auctions as a generalization of online bipartite matching [13]. There are n bidders, where each bidder i $(1 \leq i \leq n)$ has a known daily budget $B(i)$. Ad-auctions, or *products*, arrive one-by-one in an online

⋆ Work done while visiting Microsoft Research, Redmond, WA.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 253–264, 2007.

fashion. Upon arrival of a product, each buyer provides a bid $b(i,j)$ for buying it. The algorithm (i.e., the *seller*) then allocates the product to one of the interested buyers and this decision is irrevocable. The goal of the seller is to maximize the total revenue accrued. Mehta *et al.* [15] proposed a deterministic $(1-1/e)$-competitive algorithm for the case where the budget of each bidder is relatively large compared to the bids. This assumption is indeed realistic in the ad-auctions scenario.

1.1 Results and Techniques

We propose a simple algorithm and analysis for the online ad-auctions problem which is based on a clean primal-dual framework. The competitive ratio of our algorithm is $(1-1/e)$, thus matching the bounds of [15]. The primal-dual method is one of the fundamental design methodologies in the areas of approximation algorithms and combinatorial optimization. Recently, Buchbinder and Naor [7,8] have further extended the primal-dual method and have shown its applicability to the design and analysis of online algorithms. We use the primal-dual method here for both making online decisions as well as for the analysis of the competitive factors. Moreover, we observe that several other classic online problems, e.g. ski rental and TCP acknowledgement [9,12], for which (optimal) $e/(e-1)$ competitive (randomized) algorithms are known, can be viewed and analyzed within the primal-dual framework, thus leading to both simpler and more general analysis. We defer the details to the full version and just sketch how to obtain the bounds. First, an $e/(e-1)$ competitive fractional solution is computed and then the solution is rounded online with no further cost, yielding an optimal randomized algorithm. This generalizes and simplifies the online framework developed in [12]. It is no coincidence that the techniques developed for the ad-auctions problem are also applicable to the ski rental and TCP acknowledgement problems; in fact, these problems are in some sense dual problems of the ad-auctions problem. Another interesting outcome of our work is a deterministic $(1-1/e)$-competitive *fractional* algorithm[1] for the online matching problem in bipartite graphs [13]. However, rounding with no loss the fractional solution to an integral solution, thus matching the bounds of [13], remains a challenging open problem.

We remark that in [7,8] a primal-dual framework for online packing and covering problems is presented. This framework includes, for example, a large number of routing and load balancing problems [4,3,10,8], the online set cover problem [1], as well as other problems. However, in these works only logarithmic competitive factors are achieved (which are optimal in the considered settings), while the ad-auctions problem requires much more delicate algorithms and analysis. Our analysis of the algorithms we design in this paper is very simple and uses weak duality rather than a tailor made (i.e., problem specific) potential function. We believe our results further our understanding of the primal-dual method for online algorithms and we are confident that the method will prove useful in other online scenarios as well.

Extensions. The $(1-1/e)$ competitive factor is tight for the general ad-auctions model considered by [15]. Therefore, obtaining improved competitive factors requires extending the model by relaxing certain aspects of it. The relaxations we study reveal the

[1] In fact, we show that the bound is slightly better as a function of the maximum degree.

flexibility of the primal-dual approach, thus allowing us to derive improved bounds. The algorithms developed for the different extensions (except for the bounded degree case) build very nicely on the basic ad-auctions algorithm, thus allowing us to gain more insight into the primal-dual method. We also believe that the extensions we consider result in more realistic ad-auctions models. We consider four relaxations and extensions of the basic model.

Multiple Slots. Typically, in search engines, keywords can be allocated to several advertisement slots. A slot can have several desired properties for a specific buyer, such as rank in the list of ads, size, shape, etc. We extend the basic ad-auctions algorithm to a scenario in which there are ℓ slots to which ad-auctions can be allocated. Buyers are allowed to provide *slot dependent* bids on keywords and we assume that each buyer would like to buy only a *single* slot in each round. Our basic algorithm generalizes very easily to handle this extension, yielding a competitive factor of $1 - 1/e$. Specifically, the algorithm computes in each round a maximum weight matching in a bipartite graph of slots and buyers. The proof then uses the fact that there exists an optimal primal-dual solution to the (integral) matching problem. In retrospective, our basic ad-auctions algorithm can be viewed as computing a maximum weight matching in a (degenerate) bipartite graph in which one side contains a single vertex/slot. We note that Mehta et al. [15] also considered a multiple slots setting, but with the restriction that each bidder has the same bid for all the slots.

Incorporating Stochastic information. Suppose that it is known that a bidder is likely to spend a good fraction of its daily budget. This assumption is justified either stochastically or by experience. We want to tweak the basic allocation algorithm so that the worst case performance improves. As we tweak the algorithm it is likely that the bidder spends either a smaller or a larger fraction of its budget. Thus, we propose to tweak the algorithm gradually until a steady state is reached, i.e., no more tweaking is required. Suppose that at the steady state bidder i is likely to spent g_i fraction of its budget. In a realistic modeling of a search engine it is likely to assume that the number of times each query appears each day is more or less the same. Thus, no matter what is the exact keyword pattern, each of the advertisers spends a good fraction of its budget, say 20%. This allows us to improve the worst case competitive ratio of our basic ad-auctions algorithm. In particular, when the ratio between the bid and the budgets is small, the competitive ratio improves from $1 - 1/e$ to $1 - \frac{1-g}{e^{1-g}}$, where $g = \min_{i \in I}\{g_i\}$ is the minimum fraction of budget extracted from a buyer. As expected, the worst case competitive ratio is $(1 - 1/e)$ when $g = 0$, and it is 1 when $g = 1$.

Bounded Degree Setting. The proof of the $(1 - 1/e)$ lower bound on the competitiveness in [15] uses the fact that the number of bidders interested in a product can be unbounded and, in fact, can be as large as the total number of bidders. This assumption may not be realistic in many settings. In particular, the number of bidders interested in buying an ad for a specific query result is typically small (for most ad-auctions). Therefore, it is interesting to consider an online setting in which, for each product, the number of bidders interested in it is at most $d \ll n$. The question is whether one can take advantage of this assumption and design online algorithms with better competitive factor (better than $1 - 1/e$) in this case.

	Lower Bound	Upper Bound		Lower Bound	Upper Bound
$d = 2$	0.75	0.75	$d = 10$	0.662	0.651
$d = 3$	0.704	0.704	$d = 20$	0.648	0.641
$d = 5$	0.686	0.672	$d \to \infty$	0.6321...	0.6321 ...

Fig. 1. Summary of upper and lower bounds on the competitive ratio for certain values of d

As a first step, we resolve this question positively in a slightly simpler setting, which we call the *allocation problem*. In the allocation problem, the seller introduces the products one-by-one and sets a fixed price $b(j)$ for each product j. Upon arrival of a product, each buyer announces whether it is interested in buying it for the set price and the seller decides (instantly) to which of the interested buyers to sell the product. We have indications that solving the more general ad-auctions problem requires overcoming a few additional obstacles. Nevertheless, achieving better competitive factors for the allocation problem is a necessary non-trivial step. We design an online algorithm with competitive ratio $C(d) = 1 - \frac{d-1}{d(1+\frac{1}{d-1})^{d-1}}$. This factor is strictly better than $1 - 1/e$ for any value of d, and approaches $(1 - 1/e)$ from above as d goes to infinity. We also prove lower bounds for the problem that indicate that the competitive factor of our online algorithm is quite tight. Our improved bounds for certain values of d are shown in Figure 1.

Our improved competitive factors are obtained via a new approach. Our algorithm is composed of two conceptually separate phases that run simultaneously. The first phase generates online a fractional solution for the problem. A fractional solution for the problem allows the algorithm to sell each product in fractions to several buyers. This problem has a motivation of its own in case products can be divided between buyers. An example of a divisible product is the allocation of bandwidth in a communication network. This part of our algorithm that generates a fractional solution in an online fashion is somewhat counter-intuitive. In particular, a newly arrived product is not split equally between buyers who have spent the least fraction of their budget. Such an algorithm is referred to as a "water level" algorithm and it is not hard to verify that it does not improve upon the $(1-1/e)$ worst case ratio, even for small values of d. Rather, the idea is to split the product between several buyers that have **approximately** spent the same fraction of their total budget. The analysis is performed through (online) linear programming *dual fitting*: we maintain during each step of the online algorithm a dual fractional solution that bounds the optimum solution from above. We also remark that this part of the algorithm yields a competitive solution even when the prices of the products are large compared with the budgets of the buyers. As a special case, the first phase implies a $C(d)$-competitive algorithm for the online maximum fractional matching problem in bounded degree bipartite graphs [13].

The second phase consists of rounding the fractional solution (obtained in the first phase) in an online fashion. We note again that this is only a conceptual phase which is simultaneously implemented with the previous phase. This step can be easily done by using randomized rounding. However, we show how to perform the rounding deterministically by constructing a suitable potential function. The potential function is inspired by the pessimistic estimator used to derandomize the offline problem. We show that if the price of each product is small compared with the total budget of the buyer, then

this rounding phase only reduces the revenue by a factor of $1 - o(1)$ compared to the revenue of the fractional solution.

Risk Management. Some researchers working in the area of ad-auctions argue that typically budgets are not strict. The reason they give is that if clicks are profitable, i.e., the bidder is expected to make more money on a click than the bid on the click, then why would a bidder want to limit its profit. Indeed, Google's Adwords program allows budget flexibility, e.g., it can overspend the budget by 20%. In fact, the arguments against daily budgets are valid for any investment choice. For example, if you consider investing ten thousand dollars in stock A and ten thousand dollars in stock B, then the expected gain for investing twenty thousand dollars in either stocks is not going to be less profitable in expectation (estimated with whatever means). Still, the common wisdom is to diversify and the reason is *risk management*. For example, a risk management tools may suggest that if a stock reaches a certain level, then execute buy/sell of this stock and/or buy/sell the corresponding call/put options.

Industry leaders are proposing risk management for ad-auctions too. The simplest form of risk management is to limit the investment. This gives us the notion of a *budget*. We consider a more complex form of real time risk management. Instead of strict budgets, we allow a bidder to specify how aggressive it wants to bid. For example, a bidder may specify that it wants to bid aggressively for the first hundred dollars of its budget. After having spent one hundred dollars, it still wants to buy ad-auctions if it gets them at, say, half of its bid. In general, a bidder has a monotonically decreasing function f of the budget spent so far specifying how aggressive it wants to bid. We normalize $f(0) = 1$, i.e., at the zero spending level the bidder is fully aggressive. If it has spent x dollars, then its next bid is scaled by a factor of $f(x)$. In Section 6 we show how to extend the primal-dual algorithm to deal with a more general scenario of real time risk management. For certain settings we also obtain better competitive factors.

1.2 Comparison to Previous Results

Maximizing the revenue of a seller in both offline and online settings has been studied extensively in many different models, e.g., [15,2,14,6,5]. The work of [15] builds on online bipartite matching [13] and online b-matching [11]. The online b-matching problem is a special case of the online ad-auctions problem in which all buyers have a budget of b, and the bids are either 0 or 1. In [11] a deterministic algorithm is given for b-matching with competitive ratio tending to $(1 - 1/e)$ (from below) as b grows.

The idea of designing online algorithms that first generate a fractional solution and then round it in an online fashion appeared implicitly in [1]. An explicit use of this idea, along with a general scheme for generating competitive online fractional solutions for packing and covering problems, appeared in [7]. Further work on primal-dual online algorithms appears in [8].

2 Preliminaries

In the online ad-auctions problem there is a set I of n buyers, each buyer i ($1 \leq i \leq n$) has a known daily budget of $B(i)$. We consider an online setting in which m products

Dual (Packing)		Primal (Covering)	
Maximize:	$\sum_{j=1}^{m}\sum_{i=1}^{n} b(i,j)y(i,j)$	Minimize :	$\sum_{i=1}^{n} B(i)x(i) + \sum_{j=1}^{m} z(j)$
Subject to:		Subject to:	
For each $1 \le j \le m$:	$\sum_{i=1}^{n} y(i,j) \le 1$	For each (i,j):	$b(i,j)x(i) + z(j) \ge b(i,j)$
For each $1 \le i \le n$:	$\sum_{j=1}^{m} b(i,j)y(i,j) \le B(i)$	For each i,j:	$x(i), z(j) \ge 0$
For each i,j:	$y(i,j) \ge 0$		

Fig. 2. The fractional ad-auctions problem (the dual) and the corresponding primal problem

arrive one-by-one in an online fashion. Let M denote the set of all the products. The bid of buyer i on product j (which states the amount of money it is willing to pay for the item) is $b(i,j)$. The online algorithm can *allocate* (or *sell*) the product to any one of the buyers. We distinguish between *integral* and *fractional* allocations. In an integral allocation, a product can only be allocated to a single buyer. In a fractional allocation, products can be fractionally allocated to several buyers, however, for each product, the sum of the fractions allocated to buyers cannot exceed 1. The revenue received from each buyer is defined to be the minimum between the sum of the costs of the products allocated to a buyer (times the fraction allocated) and the total budget of the buyer. That is, buyers can never be charged by more than their total budget. The objective is to maximize the total revenue of the seller. Let $R_{\max} = \max_{i\in I, j\in M}\{\frac{b(i,j)}{B(i)}\}$ be the maximum ratio between a bid of any buyer and its total budget.

A linear programming formulation of the fractional (offline) ad-auctions problem appears in Figure 2. Let $y(i,j)$ denote the fraction of product j allocated to buyer i. The objective function is maximizing the total revenue. The first set of constraints guarantees that the sum of the fractions of each product is at most 1. The second set of constraints guarantees that each buyer does not spend more than its budget. In the primal problem there is a variable $x(i)$ for each buyer i and a variable $z(j)$ for each product j. For all pairs (i,j) the constraint $b(i,j)x(i) + z(j) \ge b(i,j)$ needs to be satisfied.

3 The Basic Primal-Dual Online Algorithm

The basic algorithm for the online ad-auctions produces primal and dual solutions to the linear programs in Figure 2.

Allocation Algorithm: Initially $\forall i \;\; x(i) \leftarrow 0$.
Upon arrival of a new product j allocate the product to the buyer i that maximizes $b(i,j)(1-x(i))$. If $x(i) \ge 1$ then do nothing. Otherwise:

1. Charge the buyer the minimum between $b(i,j)$ and its remaining budget and set $y(i,j) \leftarrow 1$
2. $z(j) \leftarrow b(i,j)(1-x(i))$
3. $x(i) \leftarrow x(i)\left(1+\frac{b(i,j)}{B(i)}\right) + \frac{b(i,j)}{(c-1)\cdot B(i)}$ (c is determined later).

The intuition behind the algorithm is the following. If the competitive ratio we are aiming for is $1-1/c$, then we need to guarantee that in each iteration the change in the

primal cost is at most $1 + 1/(c-1)$ the change in the dual profit. The value of c is then maximized such that both the primal and the dual solutions remain feasible.

Theorem 1. *The allocation algorithm is* $(1 - 1/c)\,(1 - R_{\max})$*-competitive, where* $c = (1 + R_{\max})^{\frac{1}{R_{\max}}}$. *When* $R_{\max} \to 0$ *the competitive ratio tends to* $(1 - 1/e)$.

Proof. We prove three simple claims:

1. The algorithm produces a primal feasible solution.
2. In each iteration: (*change in primal objective function*)/ (*change in dual objective function*) $\leq 1 + \frac{1}{c-1}$.
3. The algorithm produces an almost feasible dual solution.

Proof of (1)**:** Consider a primal constraint corresponding to buyer i and product j. If $x(i) \geq 1$ then the primal constraint is satisfied. Otherwise, the algorithm allocates the product to the buyer i' for which $b(i', j)(1 - x(i'))$ is maximized. Setting $z(j) = b(i', j)(1 - x(i'))$ guarantees that the constraint is satisfied for all (i, j). Subsequent increases of the variables $x(i)$'s cannot make the solution infeasible.

Proof of (2)**:** Whenever the algorithm updates the primal and dual solutions, the change in the dual profit is $b(i, j)$. (Note that even if the remaining budget of buyer i to which product j is allocated is less than its bid $b(i, j)$, variable $y(i, j)$ is still set to 1.) The change in the primal cost is:

$$B(i)\Delta x(i) + z(j) = \left(b(i,j)x(i) + \frac{b(i,j)}{c-1}\right) + b(i,j)(1 - x(i)) = b(i,j)\left(1 + \frac{1}{c-1}\right).$$

Proof of (3)**:** The algorithm never updates the dual solution for buyers satisfying $x(i) \geq 1$. We prove that for any buyer i, when $\sum_{j \in M} b(i, j)y(i, j) \geq B(i)$, then $x(i) \geq 1$. This is done by proving that:

$$x(i) \geq \frac{1}{c-1}\left(c^{\frac{\sum_{j \in M} b(i,j)y(i,j)}{B(i)}} - 1\right). \tag{1}$$

Thus, whenever $\sum_{j \in M} b(i, j)y(i, j) \geq B(i)$, we get that $x(i) \geq 1$. We prove (1) by induction on the (relevant) iterations of the algorithm. Initially, this assumption is trivially true. We are only concerned with iterations in which a product, say k, is sold to buyer i. In such an iteration we get that:

$$\begin{aligned}
x(i)_{\text{end}} &= x(i)_{\text{start}} \cdot \left(1 + \frac{b(i,k)}{B(i)}\right) + \frac{b(i,k)}{(c-1)\cdot B(i)} \\
&\geq \frac{1}{c-1}\left[c^{\frac{\sum_{j \in M \setminus \{k\}} b(i,j)y(i,j)}{B(i)}} - 1\right] \cdot \left(1 + \frac{b(i,k)}{B(i)}\right) + \frac{b(i,k)}{(c-1)\cdot B(i)} \qquad (2) \\
&= \frac{1}{c-1}\left[c^{\frac{\sum_{j \in M \setminus \{k\}} b(i,j)y(i,j)}{B(i)}} \cdot \left(1 + \frac{b(i,k)}{B(i)}\right) - 1\right] \\
&\geq \frac{1}{c-1}\left[c^{\frac{\sum_{j \in M \setminus \{k\}} b(i,j)y(i,j)}{B(i)}} \cdot c^{\left(\frac{b(i,k)}{B(i)}\right)} - 1\right] = \frac{1}{c-1}\left[c^{\frac{\sum_{j \in M} b(i,j)y(i,j)}{B(i)}} - 1\right] \qquad (3)
\end{aligned}$$

where Inequality (2) follows from the induction hypothesis, and Inequality (3) follows since, for any $0 \leq x \leq y \leq 1$, $\frac{\ln(1+x)}{x} \geq \frac{\ln(1+y)}{y}$. Note that when $\frac{b(i,k)}{B(i)} = R_{\max}$ then Inequality 3 holds with equality. This is the reason why we chose the value c to be $(1+R_{\max})^{\frac{1}{R_{\max}}}$. Thus, it follows that whenever the sum of charges to a buyer exceeds the budget, we stop charging this buyer. Hence, there can be at most one iteration in which a buyer is charged by less than $b(i,j)$. Therefore, for each buyer i: $\sum_{j \in M} b(i,j)y(i,j) \leq B(i) + \max_{j \in M}\{b(i,j)\}$, and thus the profit extracted from buyer i is at least:

$$\left[\sum_{j \in M} b(i,j)y(i,j)\right] \frac{B(i)}{B(i) + \max_{j \in M}\{b(i,j)\}} \geq \left[\sum_{j \in M} b(i,j)y(i,j)\right](1 - R_{\max}).$$

By the second claim the dual it at least $1 - 1/c$ times the primal, and thus (by weak duality) we conclude that the competitive ratio of the algorithm is $(1-1/c)\,(1 - R_{\max})$.

3.1 Multiple Slots

In this section we show how to extend the algorithm in a very elegant way to sell different advertisement slots in each round. Suppose there are ℓ slots to which ad-auctions can be allocated and suppose that buyers are allowed to provide bids on keywords which are slot dependent. Denote the bid of buyer i on keyword j and slot k by $b(i,j,k)$. The restriction is that an (integral) allocation of a keyword to two different slots cannot be sold to the same buyer. The linear programming formulation of the problem is in Figure 3. Note that the algorithm does not update the variables $z(\cdot)$ and $s(\cdot)$ explicitly. These variables are only used for the purpose of analysis, and are updated conceptually in the proof using the strong duality theorem. The algorithm for the online ad-auctions problem is as follows.

Allocation Algorithm: Initially, $\forall i,\ \ x(i) \leftarrow 0$. Upon arrival of a new product j:

1. Generate a bipartite graph H: n buyers on one side and ℓ slots on the other side. Edge $(i,k) \in H$ has weight $b(i,j,k)(1 - x(i))$.
2. Find a maximum weight (integral) matching in H, i.e., an assignment to the variables $y(i,j,k)$.
3. Charge buyer i the minimum between $\sum_{k=1}^{\ell} b(i,j,k)y(i,j,k)$ and its remaining budget.
4. For each buyer i, if there exists slot k for which $y(i,j,k) > 0$:

$$x(i) \leftarrow x(i)\left(1 + \frac{b(i,j,k)y(i,j,k)}{B(i)}\right) + \frac{b(i,j,k)y(i,j,k)}{(c-1)\cdot B(i)}$$

Theorem 2. *The algorithm is $(1-1/c)\,(1-R_{\max})$-competitive, where c tends to e when $R_{\max} \to 0$.*

4 Incorporating Stochastic Information

In this Section we improve the worst case competitive ratio when additional stochastic information is available. We assume that stochastically or with historical experience we

Dual (Packing)	
Maximize:	$\sum_{j=1}^{m}\sum_{i=1}^{n}\sum_{\ell=1}^{k} b(i,j,\ell)y(i,j,\ell)$
Subject to:	
$\forall 1 \le j \le m, 1 \le k \le \ell$:	$\sum_{i=1}^{n} y(i,j,k) \le 1$
$\forall 1 \le i \le n$:	$\sum_{j=1}^{m}\sum_{k=1}^{\ell} b(i,j,k)y(i,j,k) \le B(i)$
$\forall 1 \le j \le m, 1 \le i \le n$:	$\sum_{k=1}^{\ell} y(i,j,k) \le 1$

Primal (Covering)	
Minimize :	$\sum_{i=1}^{n} B(i)x(i) + \sum_{j=1}^{m}\sum_{k=1}^{\ell} z(j,k) + \sum_{i=1}^{n}\sum_{j=1}^{m} s(i,j)$
Subject to:	
$\forall i,j,k$:	$b(i,j,k)x(i) + z(j,k) + s(i,j) \ge b(i,j,k)$

Fig. 3. The fractional multi-slot problem (the dual) and the corresponding primal problem

know that, a bidder i is likely to spent a good fraction of her budget. We want to tweak the algorithm so that the algorithm's worst case performance improves. As we tweak the algorithm it is likely that the bidder may spent more or less fraction of her budget. So we propose to tweak the algorithm gradually until some steady state is reached, i.e., no more tweaking is required. Suppose at the steady state, buyer i is likely to spent a good fraction of his budget. Let $0 \le g_i \le 1$ be a lower bound on the fraction of the budget buyer i is going to spend. We show that having this additional information allows us to improve the worst case competitive ratio to $1 - \frac{1-g}{e^{1-g}}$, where $g = \min_{i \in I}\{g_i\}$ is the minimal fraction of budget extracted from a buyer.

The main idea behind the algorithm is that if a buyer is known to spend at least g_i fraction of his budget, then it means that the corresponding primal variable $x(i)$ will be large at the end. Thus, in order to make the primal constraint feasible, the value of $z(j)$ can be made smaller. This, in turn, gives us additional "money" that can be used to increase $x(i)$ faster. The tradeoff we have is on the value that $x(i)$ is going to be once the buyer spent g_i fraction of his budget. This value is denoted by $x_s(i)$ and we choose it so that after the buyer has spent g_i fraction of its budget, $x(i) = x_s(i)$, and after having extracting all its budget, $x(i) = 1$. In addition, we need the change in the primal cost to be the same with respect to the dual profit in iterations where we sell the product to a buyer i who has not yet spent the threshold of g_i of his budget. The optimal choice of $x_s(i)$ turns out to be $\frac{g_i}{c^{1-g_i}-(1-g_i)}$, and the growth function of the primal variable $x(i)$, as a function of the fraction of the budget spent, should be linear until the buyer has spent a g_i fraction of his budget, and exponential from that point on. The modified algorithm is the following:

Allocation Algorithm: Initially $\forall i \;\; x(i) \leftarrow 0$. Upon arrival of a new product j Allocate the product to the buyer i that maximizes $b(i,j)(1 - \max\{x(i), x_s(i)\})$, where $x_s(i) = \frac{g_i}{c^{1-g_i}-(1-g_i)}$. If $x(i) \ge 1$ then do nothing. Otherwise:

1. Charge the buyer the minimum between $b(i,j)$ and its remaining budget
2. $z(j) \leftarrow b(i,j)(1 - \max\{x(i), x_s(i)\})$
3. $x(i) \leftarrow x(i) + \max\{x(i), x_s(i)\}\frac{b(i,j)}{B(i)} + \frac{b(i,j)}{B(i)}\frac{1-g_i}{c^{1-g_i}-(1-g_i)}$ (c is determined later).

Theorem 3. *If each buyer spends at least g_i fraction of its budget, then the algorithm is: $\left(1-\frac{1-g}{c^{1-g}}\right)(1-R_{\max})$-competitive, where $c=(1+R_{\max})^{\frac{1}{R_{\max}}}$.*

5 Bounded Degree Setting

In this section we improve on the competitive ratio under the assumption that the number of buyers interested in each product is small compared with the total number of buyers. To do so, we design a modified primal-dual based algorithm. The algorithm only works in the case of a simpler setting (which is still of interest) called the *allocation problem*. Still, this construction turns out to be non-trivial and gives us additional useful insight into the primal-dual approach. In the allocation problem, a seller is interested in selling products to a group of buyers, where buyer i has budget $B(i)$. The seller introduces the products one-by-one and sets a fixed price $b(j)$ for each product j. Each buyer then announces to the seller (upon arrival of a product) whether it is interested in buying the current product for the set price. The seller then decides (instantly) to which of the interested buyers to sell the product. For each product j let $S(j)$ be the set of interested buyers. We assume that there exists an upper bound d such that for each product j, $|S(j)| \leq d$.

The main idea is to divide the buyers into *levels* according to the fraction of the budget that they have spent. For $0 \leq k \leq d$, let $L(k)$ be the set of buyers that have spent at least a fraction of $\frac{k}{d}$ and less than a fraction of $\frac{k+1}{d}$ of their budget (buyers in level d exhausted their budget). We refer to each $L(k)$ as level k and say that it is *nonempty* if it contains buyers. We design an algorithm for the online allocation problem using two conceptual steps. First, we design an algorithm that is allowed to allocate each product in fractions. We bound the competitive ratio of this algorithm with respect to the optimal fractional solution for the problem. We then show how to deterministically produce an integral solution that allocates each product to a single buyer. We prove that when the prices of the products are small compared to the total budget, the loss of revenue in this step is at most an $o(1)$ with respect to the fractional solution.

Our fractional allocation algorithm is somewhat counter-intuitive. In particular, the product is not split equally between buyers that spent the least fraction of their budget[2], but rather to several buyers that have approximately spent the same fraction of their total budget. The formal description of the algorithm is the following:

Allocation Algorithm: Upon arrival of a new product j allocate the product to the buyers according to the following rules:

- Allocate the product equally and continuously between interested buyers in the lowest non empty level that contain buyers from $S(j)$. If during the allocation some of the buyers moved to a higher level, then continue to allocate the product equally only among the buyers in the lowest level.
- If all interested buyers in the lowest level moved to a higher level, then allocate the remaining fraction of the product equally and continuously between the buyers in the new lowest level. If all interested buyers have exhausted their budget, then stop allocating the remaining fraction of the product.

[2] Such an algorithm is usually called a "water level" algorithm.

The idea of the analysis is to find the best tradeoff function, f_d, that relates the value of each primal variable to the value of its corresponding dual constraint. It turns out that the best function is a piecewise linear function that consists of d linear segments. As d grows, the function approximates the exponential function $f_{(d=\infty)}(x) = \frac{e^x-1}{e-1}$. Due to lack of space we defer the full analysis to the full version of the paper and just state the main theorem we proved.

Theorem 4. *The allocation algorithm is $C(d)$-competitive with respect to the optimal offline fractional solution, where: $C(d) = 1 - \frac{d-1}{d\left(1+\frac{1}{d-1}\right)^{d-1}}$.*

As stated we prove that when the prices of the products are small compared to the total budget, we may transform the fractional allocation to an integral allocation with only a small loss of revenue. We do so, by introducing another potential function that is used to make the rounding decisions. We defer the description of this part to the full version of the paper.

Lower Bounds. As we stated earlier, the standard lower bound example makes use of products with large number of interested buyers. Though, the same example when restricted to bounded degree d gives quite tight bounds. Inspecting this bound more accurately we can prove the following lower bound for any value d. Figure 1 provides the lower bounds for several sample values of d.

Lemma 1. *For any value d: $C(d) \leq 1 - \frac{k-kH(d)+\sum_{i=1}^{k} H(d-i)}{d}$, where $H(\cdot)$ is the harmonic number, and k is the largest value for which $H(d) - H(d-k) \leq 1$.*

Our bound is only tight for $d = 2$. We can derive better tailor-made lower bounds for specific values of d. In particular it is not hard to show that our algorithm is optimal for $d = 3$. We defer this result to the full version of the paper.

6 Risk Management

We extend our basic ad-auctions algorithm to handle a more general setting of real time risk management. Here each buyer has a monotonically decreasing function f of budget spent, specifying how aggressive it wants to bid. We normalize $f(0) = 1$, i.e., at the zero spending level it is fully aggressive. If it has spent x dollars then its next bid is scaled by a factor of $f(x)$. Note that since f is a monotonically decreasing function, the revenue obtained by allocating buyer i a set of items is a concave function of $\sum_j b(i,j)y(i,j)$.

Since we are interested in solving this problem integrally we assume the revenue function is piecewise linear. Let R_i be the revenue function of buyer i. Let r_i be the number of pieces of the function R_i. We define for each buyer $(r_i - 1)$ different budgets $B(i,r)$, defining the amount of money spent in each "aggression" level. When the buyer spends money from budget $B(i,r)$, the aggression ratio is $a(i,r) \leq 1$ ($a(i,1) = 1$ for each buyer i). We then define a new linear program with variables $y(i,j,k)$ indicating that item j is sold to buyer i using the kth budget. Note that the ad-auctions problem considered earlier is actually this generalized problem with two pieces. In some scenarios it is likely to assume that each buyer has a lowest "aggression" level that is strictly more than zero. For instance, a buyer is always willing to buy an item

if he only needs to pay 10% of its value (as estimated by the buyer's bid). Our modified algorithm for this more general setting takes advantage of this fact to improve the worst case competitive ratio. In particular, let $a_{\min} = \min_{i=1}^{n}\{a(i, r_i)\}$ be the minimum "aggression" level of the buyers, then the competitive factor of the algorithm is $\frac{e-1}{e-a_{\min}}$. If the minimum level is only 10% (0.1), for example, the competitive ratio is 0.656, compared with $0.632 \approx 1 - 1/e$ of the basic ad-auctions algorithm. Let $R_{\max} = \max_{i\in I, j\in M, 1\le r\le r_i-1}\{\frac{a(i,r)b(i,j)}{B(i,r)}\}$ be the maximum ratio between a charge to a budget and the total budget. The modified linear program for the more general risk management setting along with our modified algorithm are deferred to the full version.

Theorem 5. *The algorithm is* $\left(\frac{c-1}{c-a_{\min}}\right)(1 - R_{\max})$*-competitive, where* c *tends to* e *when* $R_{\max} \to 0$.

Acknowledgements. We thank Allan Borodin for pointing to us the mutiple slot setting.

References

1. Alon, N., Awerbuch, B., Azar, Y., Buchbinder, N., Naor, J.: The online set cover problem. In: Proceedings of the 35th STOC, pp. 100–105 (2003)
2. Andelman, N., Mansour, Y.: Auctions with budget constraints. In: Proc. of the 9th Scandinavian Workshop on Algorithm Theory, pp. 26–38 (2004)
3. Aspnes, J., Azar, Y., Fiat, A., Plotkin, S., Waarts, O.: On-line routing of virtual circuits with applications to load balancing and machine scheduling. J. ACM 44(3), 486–504 (1997)
4. Awerbuch, B., Azar, Y., Plotkin, S.: Throughput-competitive online routing. In: Proc. of the 34th Annual Symposium on Foundations of Computer Science, pp. 32–40 (1993)
5. Blum, A., Hartline, J.: Near-optimal online auctions (2005)
6. Borgs, C., Chayes, J., Immorlica, N., Mahdian, M., Saberi, A.: Multi-unit auctions with budget-constrained bidders. In: Proc. of the 6th EC, pp. 44–51 (2005)
7. Buchbinder, N., Naor, J.: Online primal-dual algorithms for covering and packing problems. In: Proc. of the 13th Annual European Symposium on Algorithms, pp. 689–701 (2005)
8. Buchbinder, N., Naor, J.: A primal-dual approach to online routing and packing. In: Proc. of the 47th Annual Symposium on Foundations of Computer Science, pp. 293–204 (2006)
9. Dooly, D.R., Goldman, S., Scott, S.D.: On-line analysis of the TCP acknowledgement delay problem. Journal of the ACM 48, 243–273 (2001)
10. Goel, A., Meyerson, A., Plotkin, S.A.: Combining fairness with throughput: Online routing with multiple objectives. J. Comput. Syst. Sci. 63(1), 62–79 (2001)
11. Kalyanasundaram, B., Pruhs, K.R.: An optimal deterministic algorithm for online b - matching. Theoretical Computer Science 233(1-2), 319–325 (2000)
12. Karlin, A.R., Kenyon, C., Randall, D.: Dynamic TCP acknowledgement and other stories about e/(e-1). In: Proc. of the 33rd Symposium on Theory of Computing, pp. 502–509 (2001)
13. Karp, R., Vazirani, U., Vazirani, V.: An optimal algorithm for online bipartite matching. In: Proceedings of the 22nd Annual Symposium on Theory of Computing, pp. 352–358 (1990)
14. Mahdian, M., Saberi, A.: Multi-unit auctions with unknown supply. In: EC '06: Proceedings of the 7th ACM conference on Electronic commerce, pp. 243–249 (2006)
15. Mehta, A., Saberi, A., Vazirani, U., Vazirani, V.: Adwords and generalized on-line matching. In: Proc. of the 46th IEEE Symp. on Foundations of Computer Science, pp. 264–273 (2005)
16. Robinson, S.: Computer scientists optimize innovative ad auction. SIAM News 38, 243–273 (2005)

Unique Lowest Common Ancestors in Dags Are Almost as Easy as Matrix Multiplication

Mirosław Kowaluk[1,⋆] and Andrzej Lingas[2,⋆⋆]

[1] Institute of Informatics, Warsaw University, Warsaw
kowaluk@mimum.edu.pl
[2] Department of Computer Science, Lund University, S-221 00 Lund
Andrzej.Lingas@cs.lth.se

Abstract. We consider the problem of determining for each pair of vertices of a directed acyclic graph (dag) on n vertices whether or not it has a unique lowest common ancestor, and if so, finding such an ancestor. We show that this problem can be solved in time $O(n^{\omega}\log n)$, where $\omega < 2.376$ is the exponent of the fastest known algorithm for multiplication of two $n \times n$ matrices.

We show also that the problem of determining a lowest common ancestor for each pair of vertices of an arbitrary dag on n vertices is solvable in time [1] $\widetilde{O}(n^2p+n^{\omega})$, where p is the minimum number of directed paths covering the vertices of the dag. With the help of random bits, we can solve the latter problem in time $\widetilde{O}(n^2p)$.

1 Introduction

A *lowest common ancestor* (lca) of vertices u and v in a *directed acyclic graph* (dag) is an ancestor of both u and v that has no descendant which is an ancestor of u and v, see Fig. 1 for an example. The problem of finding an lca for a pair of vertices in a (rooted) tree, or more generally, in a dag is a basic problem in algorithmic graph theory. Its numerous applications range from computing maximum matching in graphs through various string problems, inheritance analysis in programming languages and analysis of genealogical data to lattice operations in complex systems (for more details, see, e.g., [4,10,20,23]). It has been extensively studied in the context of preprocessing the input tree or dag such that lca queries can be answered quickly for any pair of vertices.

For rooted trees, Harel and Tarjan [16] provided the first linear-time preprocessing sufficient to ensure lca constant-time queries. For general dags, Bender *et al.* were the first to derive a substantially sub-cubic upper time-bound (i.e., $O(n^{(3+\omega)/2}) = O(n^{2.688})$) on the preprocessing guaranteeing constant-time lca queries [4]. Recently, Kowaluk et al. [19] and Czumaj et al. [12,11] subsequently improved this upper bound to $O(n^{2.616})$ and $O(n^{2.575})$ by reduction to the problem of determining maximum witnesses for Boolean product of two $n\times n$ Boolean

⋆ Research supported by KBN grant N20600432/0806.
⋆⋆ Research supported in part by VR grant 621-2005-4806.

[1] The notation $\widetilde{O}(f(n))$ stands for $O(f(n)\log^c n)$ for some positive constant c.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 265–274, 2007.

matrices. (In [19,11], there is also given a natural $O(nm)$-time method for the all-pairs lca problem in a dag, where m is the number of edges in the input dag. This method is superior for sparse dags).

A reverse size-preserving reduction from the maximum witness problem for Boolean $n \times n$ matrix product to the all-pairs lca problem in dags on n vertices is not known. One of the difficulties here is that a maximum witness for an entry of Boolean product is unique whereas generally a pair of vertices in a dag can have several lowest common ancestors, see Fig. 1. Thus, such a reverse reduction likely would reduce the maximum witness problem to a special case of the all-pair lca problem where there is need to report lca only for those pairs which a have a unique lca (this is the case of the reverse reduction presented in [11] which unfortunately squares the parameter n).

The maximum witness problem for Boolean matrix product is of interest in its own rights, and recently Shapira et al. demonstrated that several other important problems are easily reducible or even computationally equivalent to it (up to constant factors), for example, the problem of the all-pairs bottleneck paths in vertex weighted graphs [22]. The scenario that the maximum witness problem has substantially higher time complexity than that of the all-pairs lca problem in a dag in the absence of the aforementioned reverse reduction seems possible.

In the first part of our paper, we consider the problem of determining lowest common ancestors for all pairs of vertices that have exactly one lca (for pairs that do not have unique lca a negative answer is required). We show that this problem is solvable in time $O(n^{\omega} \log n)$. By our result, a reverse (even slightly non-preserving size) reduction from the maximum witness problem for Boolean $n \times n$ matrix product to the all-pairs unique lca problem in a dag on up to about $O(n^{1.083})$ vertices would yield substantially better upper time-bound for the former problem and the aforementioned class of important problems related to it [22]. Therefore, our result can be also regarded as an evidence that the aforementioned reverse reduction from maximum witnesses to all-pairs lca is either unlikely or highly non-trivial.

The problem of detecting unique lca is also of interest in its own rights similarly as known "unique" variants of many combinatorial problems (cf. [14])[2]. For example, a solution to it allows to detect those pairs in a dag that would be strongly affected by a failure of a single vertex in the context of lca.

In the second part of the paper, we combine the reducibility of the general all-pairs lca problem in a dag to the maximum witness problem for Boolean matrix product with known fast combinatorial algorithms for Boolean matrix product in special cases [5,13] in order to derive improved upper time-bounds for the former problem in the corresponding cases. We conclude in particular that the problem of determining a lowest common ancestor for each pair of vertices of an arbitrary dag on n vertices is solvable in time $\widetilde{O}(n^2 p + n^{\omega})$, where p is the

[2] One should also recall that the idea of the fastest known algorithm for witnesses of Boolean matrix product due to Alon et al. is based on an extention of an algorithmic solution to the case where witnesses are unique [1,21].

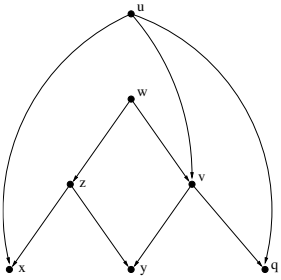


Fig. 1. A dag with 7 vertices. The lca of vertices x and y are both, vertex u and vertex z; vertex w is a common ancestor of x and y but it is not the lca of x and y. There is no common ancestor of vertices w and q. Vertex w is the unique lca of z and v while v is the unique lca of y and q.

minimum number of directed paths covering the vertices of the dag. With the help of random bits, we can solve the latter problem in time $\widetilde{O}(n^2p)$. Observe that a dag does not have to be sparse to be vertex coverable with a small number of directed paths.

Organization. Our paper is structured as follows. In Preliminaries, we introduce several concepts and facts used by our matrix based methods for the all-pairs lca problem in dags. In Section 3, we present our $O(n^\omega \log n)$-time algorithm for the all-pairs unique lca problem. In Section 3, we present our method for the general all-pairs lca problem whose time complexity can be expressed in particular in terms of the minimum number of paths covering the vertices in the input dag.

2 Preliminaries

We shall use several facts and concepts related to matrix multiplication, in particular Boolean matrix product.

Let ω denote the exponent of square matrix multiplication, that is, the smallest constant for which the product of two $n \times n$ matrices can be computed in $O(n^\omega)$ time. The best asymptotic upper bound on ω currently known is $\omega < 2.376$, by Coppersmith and Winograd [8].

The following fact which relates graph problems to fast matrix multiplication is well known.

Fact 1. *The transitive closure of any directed graph with n vertices, in particular a dag, can be computed in time $O(n^\omega)$.*

We define the concept of witness and maximum witness in Boolean matrix multiplication as follows.

Definition 1. *If an entry $C[i,j]$ of the Boolean product of two Boolean matrices A and B is equal to 1 then any index k such that $A[i,k]$ and $B[k,j]$ are equal to*

1 *is a* ***witness*** *for* $C[i,j]$. *If* k *is the largest possible witness for* $C[i,j]$ *then it is called the* ***maximum witness*** *for* $C[i,j]$. *The* ***minimum witness*** *for* $C[i,j]$ *is defined analogously.*

The following fact is due to Alon and Naor [1].

Fact 2. *The witnesses of the Boolean product of two* $n \times n$ *Boolean matrices can be computed in time* $\widetilde{O}(n^\omega)$.

The next fact is due to Czumaj, Kowaluk and Lingas [11].

Fact 3. *The maximum witnesses (or, minimum witnesses, respectively) for all positive entries of the Boolean product of two* $n \times n$ *Boolean matrices can be computed in time* $O(n^{2.575})$.

3 A Fast Algorithm for Unique lca in Dags

In this section we show that all unique lca in a dag can be determined in time $O(n^\omega \log n)$. We begin with the following theorem.

Theorem 1. *Let* G *be a dag on* n *vertices such that for each pair of its vertices having a common ancestor, there is exactly one lowest common ancestor. The all-pairs lca problem for* G *can be solved in time* $O(n^\omega \log n)$.

Proof. Let V be the vertex set of G, and let V' be a set $\{v' | v \in V\}$ of copies of the vertices in V. We shall consider auxiliary dags G' on the vertex set $V \cup V'$ which are formed from G by augmenting the set of direct ancestors for some vertices $v \in V$ with v'. The vertices in V are numbered by 1 through n whereas those in V' by $n+1$, $n+2$,... .

First, we show how to construct a dag G' such that for $i = 1, ..., n$, the i-th vertex of G, where the last bit of the binary representation of i is 1, has an odd number of ancestors in G' whereas all remaining vertices of G have an even number of ancestors in G'.

To specify G' in such a way, first we compute the transitive closure of G which takes $O(n^\omega)$ time by Fact 1. On the basis of the transitive closure, we can easily compute the number of ancestors for each vertex in G and the parity of this number. Importantly, we assume here that each vertex is its own ancestor. Next, we sort the vertices of G in topological order in time $O(n^2)$ [9], and scan them in this order performing the following operations for the currently scanned i-th vertex of G:

If the i-th vertex has an even number of ancestors in the current G' and the last bit of the binary representation of i is 1, or conversely, the i-th vertex of G has an odd number of ancestors in the current G' and the last bit of the binary representation of i is 0, then we augment the set of direct ancestors of the i-th vertex with the copy of the i-th vertex and update the parity of the set of ancestors for all descendants of the i-th vertex in G using the transitive closure of G.

Note that the updates take totally time $O(n \times n)$ time and hence the whole construction of G' takes time $O(n^\omega + n^2) = O(n^\omega)$.

Given such a dag G', we compute the transitive closure $(A')^*$ of its adjacency matrix A' in time $O(n^\omega)$. Next, we compute the product C of $(A')^*$ with its transpose in Z_{2n+1}.

Note that for $1 \leq i \leq n$ and $1 \leq k \leq 2n$, the k-th element of the i-th row of $(A')^*$ is 1 iff the k-th vertex of G' is an ancestor of the i-th vertex of G. Analogously, for $1 \leq j \leq n$ and $1 \leq k \leq 2n$, the k-th element of the j-th column of the transpose of $(A')^*$ is 1 iff the k-th vertex of G' is an ancestor of the i-th vertex of G. Hence, $C[i, j]$ is equal to the number of common ancestors of the i-th and the j-th vertex of G in G'.

By the uniqueness of the lca of the i-th and the j-th vertex in G, the lca, say the k-th vertex of G, has exactly $C[i, j]$ ancestors. Hence, the parity of k is the same as that of $C[i, j]$ by the specification of G'.

Next, for $l = 2, ..., O(\log n)$, we iterate the procedure specifying in a such way G' that the number of ancestors of the i-th vertex of G in G' has the parity of the l-th bit in the reversed binary representation of i. The whole algorithm termed as UniqueLCA is given in Fig. 2.

In effect, for each pair of vertices of G, we obtain either the full binary representation of its unique lowest ancestor or just a sequence of zeros if they do not have a common ancestor.

Since the time complexity of each iteration is dominated by the time required to find the transitive closure of G', the total running time of the method is $O(n^\omega \log n)$.

The algorithm of Theorem 1 run on an arbitrary dag (see Fig. 3 for an example) might happen to produce a candidate for a unique lca for a vertex pair that does not have a unique lca. The following theorem will be helpful in handling such a situation.

Theorem 2. *Let $G = (V, E)$ be a dag on n vertices, and let $P \subseteq V \times V$. Suppose that for each pair $(u,\ v)$ in P, there is a candidate vertex $c(u, v)$ for a unique lca for u and v in G. One can verify the correctness of all the candidate vertices in P in time $O(n^\omega)$.*

Proof. By computing the transitive closure A^* of the adjacency matrix A of G, we can verify for all $(u,\ v) \in P$ whether or not $c(u, v)$ is a common ancestor of u and v in time $O(n^\omega + |P|) = O(n^\omega)$. Let $An(v)$ denote the set of ancestors of v inclusive itself. If the candidate vertex $c(u, v)$ is a common ancestor of u and v, then $An(c(u, v))$ is a subset of the set of all common ancestors for p. Hence, the cardinality of $An(c(u, v))$ equals that of the set of all common ancestors for u and v iff $c(u, v)$ is a unique lca for u and v.

To test the aforementioned equality for a candidate vertex $c(u, v)$, which is a common ancestor of u and v, we proceed as follows. First we precompute the cardinalities of the sets $An(v)$ of ancestors for all vertices v of G by a straightforward bottom up method in time $O(|V| + |E|) = O(n^2)$. Next, we compute the product C of A^* with its transpose in time $O(n^\omega)$.

procedure *UniqueLCA*(G)

Input: a dag G on n vertices such that for any pair of vertices in G there is a unique lca;
Output: for each pair of vertices, their unique lca.

1. Compute the transitive closure of G;
2. For each vertex of G, compute the number of its ancestors;
3. Sort topologically G;
4. **for** $l = 1...\lceil \log n \rceil$ **do**
   (a) $G' \leftarrow G$;
   (b) **while** scanning vertices of G in topological order **do**
      i. **if** the scanned vertex has an even number of ancestors in G' and the l-th bit of the reverse binary representation of its number i is 1, or conversely, the i-th vertex has an odd number of ancestors in G' and the l-th bit of the reverse binary representation of i is 0 **then** augment the set of direct ancestors of the i-th vertex of G in G' with the copy of the i-th vertex and update the parity of the cardinality of the set of ancestors for all descendants of the i-th vertex in G using the transitive closure of G;
   (c) Compute the transitive closure $(A')^*$ of the adjacency matrix A' of G';
   (d) Compute the product C of $(A')^*$ with its transpose;
   (e) For each pair of $i,\ j$ of vertices of G set the l-th bit of the reversed binary representation of their unique lca to 1 if $C[i,\ j]$ is odd and to 0 otherwise.

Fig. 2. The procedure UniqueLCA

Note that the cardinality of the set of common ancestors of the pair $v,\ u$ equals $C[v, u]$. Thus, in case $c(v, u)$ is a common ancestor of (u, v), it remains to verify whether or not $C[u, v] = |An(c(u, v))|$ which takes time $O(|P|) = O(n^2)$.

As a corollary, we obtain the following strengthening of Theorem 1.

Theorem 3. *For a dag G on n vertices, for all pairs of vertices in G one can determine whether or not they have a unique lowest common ancestor, and if so, determine the unique ancestor, in total time $O(n^\omega \log n)$.*

Interestingly, Theorem 2 can be be extended to include two candidate vertices for at most two lowest common ancestors.

Remark. *Let $G = (V, E)$ be a dag on n vertices, and let $Q \in V \times V$. Suppose that for each pair p in Q, there are candidate vertices $c_1(p)$, $c_2(p)$ for a unique pair of lca for p in G. One can verify the candidate pairs in time $O(n^\omega)$.*

Proof. sketch. The proof goes along the lines of that of Theorem 2. In particular, we check first if $c_1(p)$ and $c_2(p)$ are common ancestors of p for all pairs p. If so,

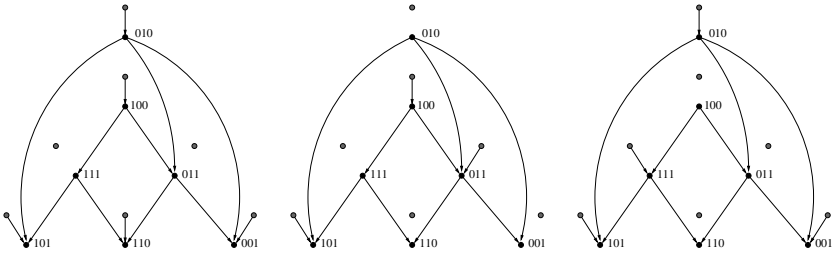


Fig. 3. Three iterations of UniqueLCA run on the example dag

then $c_1(p)$ and $c_2(p)$ form a unique pair of ancestors of p, or one of them is a unique lca of p, iff the number of common ancestors of p in G equals $|An(c_1(p))|+|An(c_2(p))|-B$, where B is the set of common ancestors of $c_1(p)$ and $c_2(p)$. Since, as in the proof of Theorem 2, the number of common ancestors for all vertex pairs in G can be computed in time $O(n^\omega)$ and the cardinalities of the sets of all ancestors for all vertices can be computed in time $O(n^2)$, we obtain the thesis.

4 Lowest Common Ancestors in Dags of Small Path Coverage

A *path coverage* of a dag is a set of directed paths which cover all vertices of the dag. A *path coverage number* of a dag G is the minimum cardinality of a path coverage of G; it is denoted by $PCN(G)$. See Fig. 3 for an example.

In this section, we show in particular that for a dag G on n vertices the all-pair lca problem can be solved deterministically in time $\widetilde{O}(n^2 PCN(G) + n^\omega)$ or by a randomized algorithm in time $\widetilde{O}(n^2 PCN(G))$.

In [5] (see also [13]), Björklund et al. have presented a combinatorial randomized algorithm for the Boolean product of two $n \times n$ matrices and the witnesses of the product which runs in time $\tilde{O}(n(n+c))$ where c is the minimum of the costs of the minimum spanning trees for the rows of A and the columns B, respectively, in the Hamming metric[3] (recall that $\tilde{O}(f(n))$ means $O(f(n)poly - \log n)$).

The algorithm of Björklund *et al.* first computes an $O(\log n)$ approximate minimum spanning tree of the rows of the first matrix in the Hamming metric as well as an $O(\log n)$ approximate minimum spanning tree of the columns of the second matrix by using a randomized algorithm of Indyk *et al.* [18]. We may assume without loss of generality that the cost of the first tree is not larger than that of the second one. Then, the algorithm fixes a traversal of the tree. Next, for each pair of consecutive neighboring rows in the traversal, it determines the

[3] A minimum spanning tree of the rows (or, columns) of a Boolean matrix in the Hamming metric is a spanning tree of the complete graph whose vertices are in one-to-one correspondance with the rows (or, columns, respectively) of the matrix and the weight of each edge is equal to the Hamming distance between the rows (or, columns, respectively) correponding to the endpoints of the edge.

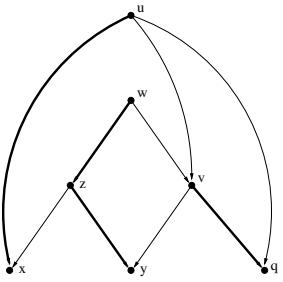


Fig. 4. An example of a minimum cardinality path coverage of a dag

positions on which they differ. Observe that the number of these positions is equal to the Hamming distance between the rows. Finally, for each column q of the second matrix, the algorithm traverses the tree and implicitly computes the set witnesses of the $\cdot$ product of the traversed row of the first matrix with q by updating the set of witnesses for the $\cdot$ product of the previously traversed row of the first matrix with q at the positions on which the two rows differ.

The algorithm of Björklund *et al.* can be easily adopted to the problem of finding maximum witnesses. Simply, the witnesses for the $\cdot$ product of the traversed row of the first matrix with the fixed column of the second one can be kept in a heap. While passing to the next row, the heap needs to be updated only for the elements corresponding to the positions on which the new row differs from the previous one. Hence, we obtain the following theorem.

Theorem 4. *There is a randomized algorithm for the maximum witnesses of the Boolean product of two $n \times n$ matrices A, B running in time $\widetilde{O}(n(n+c))$ where c is the minimum of the costs of the minimum spanning trees for the rows of A and the columns B, respectively, in the Hamming metric.*

For a dag G, let G' be the edge weighted dag obtained by assigning to each edge (v, u) the difference between the number of ancestors of u and that of v. Consider a minimum spanning forest T of G' where the roots of the trees are the sources of G'. On the other hand, consider the adjacency matrix A^* of the transitive closure of G. Let T^* be the spanning tree of the rows of A^* in the Hamming metric induced by T so that a row corresponding to a source is the root and in particular the other rows corresponding to sources are the children of the root. It is not difficult to see that T^* has the same cost as T. Recall that the maximum witnesses for the Boolean product of A^* with its transpose yield a solution to the all-pairs lca problem for G. To compute the product and its maximum witnesses we can either use the algorithm of Theorem 4 or plug in the tree T^* instead of the approximation one due to Indyk *et al.* in this algorithm. In the latter way, we can solve deterministically the all-pairs lca problem for G in time $O(n(n+t))$ where t is the cost of T provided T is given. To specify G' and compute T, we need to compute the transitive closure of G which takes $O(n^\omega)$ time. Summarizing, we obtain the following theorem.

Theorem 5. *For a dag G on n vertices, let G' be the edge weighted dag obtained by assigning to each edge (v, u) the difference between the number of ancestors of u and that of v, and let t be the minimum cost of a spanning forest of G' where the sources of G' are the roots of the trees in the forest. The all-pairs lca problem for G can be solved in time $\widetilde{O}(n(n+t)+n^{\omega})$. There is also a randomized algorithm solving the all-pairs lca problem for G in time $\widetilde{O}(n(n+t))$.*

Consider a path coverage S of a dag G. For the sake of a proof, extend each directed path in S that does not start from a source to a directed path starting from a source. It is not difficult to see that the total cost of edges in the extended paths originally in S in the weighted dag G' is not less than the cost of a minimum spanning tree of G'. On the other hand, the total cost of edges in a directed path in G' is at most n. Hence, we obtain the following theorem.

Theorem 6. *For a dag G on n vertices, the all-pairs lca problem for G can be solved in time $\widetilde{O}(n^2 PCN(G)+n^{\omega})$. There is also a randomized algorithm solving the all-pairs lca problem for G in time $\widetilde{O}(n^2 PCN(G))$.*

5 Final Remarks

Our algorithms for the unique lca problem in a dag and that for the all-pairs lca problem in an arbitrary dag whose complexity is expressed in terms of the cost of a minimum spanning tree or the path coverage number extend substantially the list of cases for which the all-pairs lca problem can solved faster than in the general case. Previously, more efficient solutions have been reported for sparse dags and bounded depth dags [11,12,19]. One can hope that by using the aforementioned more efficient cases, possibly extended by new ones, and a careful case analysis, one could substantially improve the upper bound $O(n^{2.575})$ for the general problem.

References

1. Alon, N., Naor, M.: Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. Algorithmica 16, 434–449 (1996)
2. Becker, A., Geiger, D., Schaeffer, A.A.: Automatic Selection of Loop Breakers for Genetic Linkage Analysis.
3. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–93. Springer, Heidelberg (2000)
4. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. Journal of Algorithms, 57(2), 75–94 (2005) (a preliminary version in Proc. SODA 2001, pp. 845–853)
5. Björklund, A., Lingas, A.: Fast Boolean matrix multiplication for highly clustered data. In: Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 2001. LNCS, vol. 2125, Springer, Heidelberg (2001)
6. Cole, R., Hariharan, R.: Dynamic LCA queries in trees. SIAM Journal on Computing 34(4), 894–923 (2005)

7. Coppersmith, D.: Rectangular matrix multiplication revisited. Journal of Symbolic Computation 13, 42–49 (1997)
8. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progression. Journal of Symbolic Computation 9, 251–290 (1990)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. McGraw-Hill Book Company, Boston, MA (2001)
10. Cottingham Jr., R.W., Idury, R.M., Shäffer, A.A.: Genetic linkage computations. American Journal of Human Genetics 53, 252–263 (1993)
11. Czumaj, A., Kowaluk, M., Lingas, A.: Faster algorithms for finding lowest common ancestors in directed acyclic graphs. Theoretical Computer Science 380(1-2), 37–46 (2007)
12. Czumaj, A., Lingas, A.: Improved algorithms for the all-pairs lowest common ancestor problem in directed acyclic graphs. Manuscript (2005)
13. Gasieniec, L., Lingas, A.: An improved bound on Boolean matrix multiplication for highly clustered data. In: Dehne, F., Sack, J.-R., Smid, M. (eds.) WADS 2003. LNCS, vol. 2748, pp. 329–339. Springer, Heidelberg (2003)
14. Gabow, H.N., Kaplan, H., Tarjan, R.E.: Unique Maximum Matching Algorithms. In: Proc. 31st Annual ACM Symposium on Theory of Computing (STOC'99) (1999)
15. Galil, Z., Margalit, O.: Witnesses for Boolean matrix multiplication and for transitive closure. Journal of Complexity 9, 201–221 (1993)
16. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13(2), 338–355 (1984)
17. Huang, X., Pan, V.Y.: Fast rectangular matrix multiplications and applications. Journal of Complexity 14, 257–299 (1998)
18. Indyk, P., Motwani, R.: Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In: Proc. 30th ACM Symposium on Theory of Computing (STOC'98) (1998)
19. Kowaluk, M., Lingas, A.: LCA queries in directed acyclic graphs. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 241–248. Springer, Heidelberg (2005)
20. Nykänen, M., Ukkonen, E.: Finding lowest common ancestors in arbitrarily directed trees. Information Processing Letters 50(6), 307–310 (1994)
21. Seidel, R. (ed.): On the All-Pairs-Shortest-Path Problem, pp. 745–749. ACM Press, New York (1992)
22. Shapira, A., Yuster, R., Zwick, U.: All-Pairs Bottleneck Paths in Vertex Weighted Graphs. In: Proc. 18th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA'07), pp. 978–985 (2007)
23. Shäffer, A.A., Gupta, S.K., Shriram, K., Cottingham Jr., R.W.: Avoiding recomputation in linkage analysis. Human Heredity 44, 225–237 (1994)
24. Tarjan, R.E.: Applications of path compression on balanced trees. Journal of the ACM 26(4), 690–715 (1979)

Optimal Algorithms for k-Search with Application in Option Pricing

Julian Lorenz*, Konstantinos Panagiotou**, and Angelika Steger

Institute of Theoretical Computer Science, ETH Zurich, 8092 Zurich, Switzerland
{jlorenz,panagiok,steger}@inf.ethz.ch

Abstract. In the k-search problem, a player is searching for the k highest (respectively, lowest) prices in a sequence, which is revealed to her sequentially. At each quotation, the player has to decide *immediately* whether to accept the price or not. Using the competitive ratio as a performance measure, we give optimal deterministic and randomized algorithms for both the maximization and minimization problems, and discover that the problems behave substantially different in the worst-case. As an application of our results, we use these algorithms to price "lookback options", a particular class of financial derivatives. We derive bounds for the price of these securities under a no-arbitrage assumption, and compare this to classical option pricing.

1 Introduction

1.1 k-Search Problem

We consider the following online search problem: a player wants to sell (respectively, buy) $k \geq 1$ units of an asset with the goal of maximizing her profit (minimizing her cost). At time points $i = 1, \ldots, n$, the player is presented a price quotation p_i, and must *immediately* decide whether or not to sell (buy) one unit of the asset for that price. The player is required to complete the transaction by some point in time n. We ensure that by assuming that if at time $n - j$ she has still j units left to sell (respectively, buy), she is compelled to do so in the remaining j periods. We shall refer to the profit maximization version (selling k units) as k-max-search, and to the cost minimization version (purchasing k units) as k-min-search.

In this work, we shall make no modeling assumptions on the price path except that it has finite support, which is known to the player. That is, the prices are chosen from the real interval $\mathcal{I} = \{x \,|\, m \leq x \leq M\}$, where $0 < m < M$. We define the *fluctuation ratio* $\varphi = M/m$. Let $\mathcal{P} = \bigcup_{n \geq k} \mathcal{I}^n$ be the set of all price sequences of length at least k. Moreover, the length of the sequence is known to the player at the beginning of the game.

Sleator and Tarjan [1] proposed to evaluate the performance of online algorithms by using *competitive analysis*. In this model, an online algorithm ALG is

* This work was partially supported by UBS AG.
** This work was partially supported by the SNF, grant number: 200021-107880/1.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 275–286, 2007.

compared with an offline optimum algorithm OPT (which knows all prices in advance), on the same price sequence. Here, the price sequence is chosen by an adversary out of the set $\mathcal{P}$ of admissible sequences. Let $\mathsf{ALG}(\sigma)$ and $\mathsf{OPT}(\sigma)$ denote the objective values of ALG and OPT when executed on $\sigma \in \mathcal{P}$. The *competitive ratio* of ALG is defined for maximization problems as

$$\mathrm{CR}(\mathsf{ALG}) = \max\left\{ \frac{\mathsf{OPT}(\sigma)}{\mathsf{ALG}(\sigma)} \;\middle|\; \sigma \in \mathcal{P} \right\},$$

and similarly, for minimization problems

$$\mathrm{CR}(\mathsf{ALG}) = \max\left\{ \frac{\mathsf{ALG}(\sigma)}{\mathsf{OPT}(\sigma)} \;\middle|\; \sigma \in \mathcal{P} \right\}.$$

We say that ALG is *c-competitive* if it achieves a competitive ratio not larger than c. For randomized algorithms, we substitute the *expected* objective value $\mathbb{E}[\mathsf{ALG}]$ for ALG in the definitions above.

Related Work. In 2001, El-Yaniv, Fiat, Karp and Turpin studied, among other problems, the case $k = 1$, i.e. *1-max-search*, and the closely related *one-way trading* problem [2] with the competitive ratio (defined above) as performance measure. In the latter, a player wants to exchange some initial wealth to some other asset, and is again given price quotations one-by-one. However, the player may exchange an *arbitrary fraction* of her wealth for each price. Hence, the k-max-search problem for general $k \geq 1$ can be understood as a natural bridge between the two problems considered in [2], with $k \to \infty$ corresponding to the one-way trading problem. This connection will be made more explicit later.

Several variants of search problems, which will discussed below, have been extensively studied in operations research and mathematical economics. However, traditionally most of the work follows a *Bayesian* approach: optimal algorithms are developed under the assumption that the prices are generated by a known distribution. Naturally, such algorithms heavily depend on the underlying model.

Lippmann and McCall [3,4] give an excellent survey on search problems with various assumptions on the price process. More specifically, they study the problem of job and employee search and the economics of uncertainty, which are two classical applications of series search problems. In [5], Rosenfield and Shapiro study the situation where the price follows a random process, but some of its parameters that may be random variables with known prior distribution. Hence, the work in [5] tries to get rid of the assumption of the Bayesian search models that the underlying price process is *fully* known to the player. Ajtai, Megiddo and Waarts [6] study the classical *secretary* problem. Here, n objects from an ordered set are presented in random order, and the player has to accept k of them so that the final decision about each object is made only on the basis of its rank relative to the ones already seen. They consider the problems of maximizing the probability of accepting the best k objects, or minimizing the expected sum of the ranks (or powers of ranks) of the accepted objects. In this context, Kleinberg designed in [7] an $(1 - \mathcal{O}(1/\sqrt{k}))$-competitive algorithm for the problem of maximizing the sum of the k chosen elements.

Results & Discussion. In contrast to the Bayesian approaches, El-Yaniv et al. [2] circumvent almost all distributional assumptions by resorting to competitive analysis and the minimal assumption of a known finite price interval. In this paper we also follow this approach. The goal is to provide a generic search strategy that works with any price evolution, rather than to retrench to a specific stochastic price process. In many applications, where it is not clear how the generating price process should be modeled, this provides an attractive alternative to classical Bayesian search models. In fact, in the second part of the paper we give an interesting application of k-max-search and k-min-search to *robust option pricing* in finance, where relaxing typically made assumptions on the (stochastic) price evolution to the minimal assumption of a *price interval* yields remarkably good bounds.

Before we proceed with stating our results, let us introduce some notation. For $\sigma \in \mathcal{P}, \sigma = (p_1, \ldots, p_n)$, let $p_{max}(\sigma) = \max_{1 \le i \le n} p_i$ denote the maximum price, and $p_{min}(\sigma) = \max_{1 \le i \le n} p_i$ the minimum price. Let W denote *Lambert's W-function*, i.e., the inverse of $f(w) = w \exp(w)$. For brevity we shall write $f(x) \sim g(x)$, if $\lim_{x \to \infty} f(x)/g(x) = 1$. It is well-known that $W(x) \sim \log x$.

Our results for deterministic k-max-search are summarized in Theorem 1.

Theorem 1. *Let $k \in \mathbb{N}$, $\varphi > 1$. There is a r^*-competitive deterministic algorithm for k-max-search, where $r^* = r^*(k, \varphi)$ is the unique solution of*

$$\frac{\varphi - 1}{r^* - 1} = \left(1 + \frac{r^*}{k}\right)^k, \tag{1}$$

and there exists no deterministic algorithm with smaller competitive ratio. Furthermore, it holds

(i) For fixed $k \ge 1$ and $\varphi \to \infty$, we have $r^*(k, \varphi) \sim \sqrt[k+1]{k^k \varphi}$.
(ii) For fixed $\varphi > 1$ and $k \to \infty$, we have $r^*(k, \varphi) \sim 1 + W(\frac{\varphi - 1}{e})$.

The algorithm in the theorem above is given explicitly in Section 2. Interestingly, the optimal competitive deterministic algorithm for the one-way trading problem studied in [2] has competitive ratio exactly $1 + W(\frac{\varphi - 1}{e})$ (for $n \to \infty$), which coincides with the ratio of our algorithm given by the theorem above for $k \to \infty$. Hence, k-max-search can indeed be understood as a natural bridge between the 1-max-search problem and the one-way trading problem.

For deterministic k-min-search we obtain the following statement.

Theorem 2. *Let $k \in \mathbb{N}$, $\varphi > 1$. There is a s^*-competitive deterministic algorithm for k-min-search, where $s^* = s^*(k, \varphi)$ is the unique solution of*

$$\frac{1 - 1/\varphi}{1 - 1/s^*} = \left(1 + \frac{1}{ks^*}\right)^k, \tag{2}$$

and there exists no deterministic algorithm with smaller competitive ratio. Furthermore, it holds

(i) For fixed $k \ge 1$ and $\varphi \to \infty$, we have $s^*(k, \varphi) \sim \sqrt{\frac{k+1}{2k} \varphi}$.
(ii) For fixed $\varphi > 1$ and $k \to \infty$, we have $s^*(k, \varphi) \sim (W(-\frac{\varphi - 1}{e\varphi}) + 1)^{-1}$.

The algorithm in the theorem above is also given explicitly in Section 2. Surprisingly, although one might think that k-max-search and k-min-search should behave similarly with respect to competitive analysis, Theorem 2 states that this is in fact *not* the case. Indeed, according to Theorems 1 and 2, for large φ, the best algorithm for k-max-search achieves a competitive ratio of roughly $k\sqrt[k]{\varphi}$, while the best algorithm for k-min-search is at best $\sqrt{\varphi/2}$-competitive. Similarly, when k is large, the competitive ratio of a best algorithm for k-max-search behaves like $\log\varphi$, in contrast to k-min-search, where a straightforward analysis (i.e. series expansion of the W function around its pole) shows that the best algorithm achieves a ratio of $\Theta(\sqrt{\varphi})$. Hence, algorithms for k-min-search perform in the worst-case rather poorly compared to algorithms for k-max-search.

Furthermore, we investigate the performance of *randomized* algorithms for the problems in question. In [2] the authors gave a $\mathcal{O}(\log\varphi)$-competitive randomized algorithm for 1-max-search, but did not provide a lower bound.

Theorem 3. *Let $k \in \mathbb{N}$, $\varphi > 1$. For every randomized k-max-search algorithm* RALG *we have*

$$\mathrm{CR}(\mathsf{RALG}) \geq (\ln\varphi)/2. \tag{3}$$

Furthermore, there is a $(\ln\varphi)/\ln 2$-competitive randomized algorithm.

Note that the lower bound above is independent of k, i.e., randomized algorithms cannot improve their performance when k increases. In contrast to that, by considering Theorem 1, as k grows the performance of the best *deterministic* algorithm improves, and approaches $\log\varphi$, which is only a multiplicative factor away from the best ratio that a randomized algorithm can achieve.

Our next result is about randomized algorithms for k-min-search.

Theorem 4. *Let $k \in \mathbb{N}, \varphi > 1$. For every randomized k-min-search algorithm* RALG *we have*

$$\mathrm{CR}(\mathsf{RALG}) \geq (1+\sqrt{\varphi})/2. \tag{4}$$

Again, the given lower bound is independent of k. More surprising, combined with Theorem 2, the theorem above states that for *all* $k \in \mathbb{N}$, randomization *does not* improve the performance (up to possibly a multiplicative constant) of algorithms for k-min-search, compared to deterministic algorithms. This is again a slightly unexpected difference between k-max-search and k-min-search.

1.2 Application to Robust Valuation of Lookback Options

In the second part of the paper we will use competitive k-search algorithms to derive upper bounds for the price of *lookback options*, a particular class of financial derivatives (see e.g. [8]). An option is a contract whereby the option holder has the right (but not obligation) to exercise a feature of the option contract on or before an exercise date, delivered by the other party – the writer of the option. Since the option gives the buyer a right, it will have a price that the buyer has to pay to the option writer.

The most basic type of options are European options on a stock. They give the holder the right to buy (respectively, sell) the stock on a prespecified date T (expiry date) for a prespecified price K. Besides these standard and well-understood types, there is also a plethora of options with more complex features. One type are so called *lookback options*. A lookback *call* allows the holder to buy the underlying stock at time T from the option writer at the historical minimum price observed over $[0, T]$, and a lookback *put* to sell at the historical maximum.

A fundamental question is to determine the value of an option at time $t < T$. Black and Scholes [9] studied European call and put options on non-dividend paying stocks in a seminal paper. The key argument in their derivation is a *no arbitrage condition*. Loosely speaking, an arbitrage is a zero-risk, zero net investment strategy that still generates profit. If such an opportunity came about, market participants would immediately start exploiting it, pushing prices until the arbitrage opportunity ceases to exist. Black and Scholes essentially give a dynamic trading strategy in the underlying stock by which an option writer can risklessly hedge an option position. Thus, the no arbitrage condition implies that the cost of the trading strategy must equal the price of the option to date.

In the model of Black and Scholes trading is possible continuously in time and in arbitrarily small portions of shares. Moreover, a central underlying assumption is that the stock price follows a geometric Brownian motion (see e.g. [10]), which then became the standard model for option pricing. While it certainly shows many features that fairly resemble reality, the behavior of stock prices in practice is not fully consistent with this assumption. For instance, the distribution observed for the returns of stock price processes are non-Gaussian and typically heavy-tailed [11], leading to underestimation of extreme price movements. Furthermore, in practice trading is discrete, price paths include price jumps and stock price volatility is not constant. As a response, numerous modifications of the original Black-Scholes setting have been proposed, examining different stochastic processes for the stock price (for instance [12,13,14]).

In light of the persistent difficulties of finding and formulating the "right" model for the stock price dynamic, there have also been a number of attempts to price financial instruments by *relaxing the Black-Scholes assumptions* instead. The idea is to provide *robust* bounds that work with (almost) any evolution of the stock price rather than focusing on a specific formulation of the stochastic process. In this fashion, DeMarzo, Kremer and Mansour [15] derive both upper and lower bounds for option prices in a model of bounded quadratic variation, using competitive online trading algorithms. In the mathematical finance community, Epstein and Wilmott [16] propose non-probabilistic models for pricing interest rate securities in a framework of "worst-case scenarios". Korn [17] combines the random walk assumption with a worst-case analysis to tackle optimal asset allocation under the threat of a crash.

In this spirit, using the deterministic k-search algorithms from Section 2 we derive in Section 4 upper bounds for the price of lookback calls and puts, under the assumption of *bounded stock price paths* and non-existence of arbitrage opportunities. Interestingly, the resulting bounds are remarkably good, showing similar qualitative properties and quantitative values as pricing in the standard

Black-Scholes model. Note that the assumption of a bounded stock price is indeed very minimal, since without *any* assumption about the magnitude of the stock price fluctuation in fact *no* upper bounds for the option price apply.

2 Deterministic Search

Let us consider the following *reservation price policy* RPP for k-max-search. Prior to the start of the game, we choose *reservation prices* p_i^* $(i = 1 \dots k)$. As the prices are sequentially revealed, RPP accepts the first price that is at least p_1^* and sells one unit. It then waits for the first price that is at least p_2^*, and subsequently continues with all reservation prices. RPP works through the reservation prices in a strictly sequential manner. Note that RPP may be forced to sell at the last prices of the sequence, which may be lower than the remaining reservations, to meet the constraint of completing the sale.

The proof of the lemma below generalizes the ideas presented in [2].

Lemma 1. *Let $k \in \mathbb{N}$, $\varphi > 1$. Let $r^* = r^*(k, \varphi)$ be defined as in (1). Then the reservation price policy* RPP *with reservation prices given by*

$$p_i^* = m\left[1 + (r^* - 1)\left(1 + r^*/k\right)^{i-1}\right], \tag{5}$$

satisfies $k\, p_{max}(\sigma) \le r^* \cdot \mathsf{RPP}(\sigma)$ for all $\sigma \in \mathcal{P}$. In particular, RPP *is a r^*-competitive algorithm for the k-max-search problem.*

Proof. For $0 \le j \le k$, let $\mathcal{P}_j \subseteq \mathcal{P}$ be the sets of price sequences for which RPP accepts *exactly* j prices, excluding the forced sale at the end. Then $\mathcal{P}$ is the disjoint union of the $\mathcal{P}_j$'s. To shorten notation, let us write $p_{k+1}^* = M$. Let $\varepsilon > 0$ be fixed and define the price sequences

$$\forall 0 \le i \le k \;:\quad \sigma_i \;=\; p_1^*, p_2^*, \dots, p_i^*, \underbrace{p_{i+1}^* - \varepsilon, \dots, p_{i+1}^* - \varepsilon}_{k}, \underbrace{m, m, \dots, m}_{k}.$$

Observe that as $\varepsilon \to 0$, each σ_j is a sequence yielding the worst-case ratio in $\mathcal{P}_j$, in the sense that for all $\sigma \in \mathcal{P}_j$

$$\frac{\mathsf{OPT}(\sigma)}{\mathsf{RPP}(\sigma)} \le \frac{k p_{\max}(\sigma)}{\mathsf{RPP}(\sigma)} \;\le\; \frac{k p_{j+1}^*}{\mathsf{RPP}(\sigma_j)}. \tag{6}$$

Thus, to prove the statement we show that for $0 \le j \le k$ it holds $k p_{j+1}^* \le r^* \cdot \mathsf{RPP}(\sigma_j)$. A straightforward calculation shows that for all $0 \le j \le k$

$$\sum_{i=1}^{j} p_i^* = m\left[j + k(1 - 1/r^*)\left((1 + r^*/k)^j - 1\right)\right].$$

But then we have for $\varepsilon \to 0$,

$$\forall\, 0 \le j \le k : \quad \frac{k p_{j+1}^*}{\mathsf{RPP}(\sigma_j)} = \frac{k p_{j+1}^*}{\sum_{i=1}^{j} p_i^* + (k-j)m} = r^*.$$

Thus, from (6) the r^*-competitiveness of RPP follows immediately. □

By considering the choice of reservation prices in Lemma 1, we see that in fact no deterministic algorithm will be able to do better than RPP in the worst-case.

Lemma 2. *Let $k \geq 1$, $\varphi > 1$. Then $r^*(k,\varphi)$ given by (1) is the lowest possible competitive ratio that a deterministic k-max-search algorithm can achieve.*

By combining Lemma 1 and Lemma 2 we immediately obtain the first part of Theorem 1. The proofs of Lemma 2 and the statements about the asymptotic behavior of the competitive ratio are omitted due to space restrictions.

Similarly, we can construct a reservation price policy RPP for k-min-search. Naturally, RPP is modified such that it accepts the first price *lower* than the current reservation price.

Lemma 3. *Let $k \in \mathbb{N}$, $\varphi > 1$. Let $s^* = s^*(k,\varphi)$ be defined as in (2). Then the reservation price policy* RPP *with reservation prices $p_1^* > \cdots > p_k^*$,*

$$p_i^* = M\left[1 - \left(1 - \frac{1}{s^*}\right)\left(1 + \frac{1}{ks^*}\right)^{i-1}\right], \tag{7}$$

satisfies $\mathsf{RPP}(\sigma) \leq s^*(k,\varphi) \cdot k\, p_{min}(\sigma)$, and is a $s^*(k,\varphi)$-competitive deterministic algorithm for k-min-search.

Again, no deterministic algorithm can do better than RPP in Lemma 3.

Lemma 4. *Let $k \geq 1$, $\varphi > 1$. Then $s^*(k,\varphi)$ given by (2) is the lowest possible competitive ratio that a deterministic k-min-search algorithm can achieve.*

The proofs of Lemma 3 and 4 and the asymptotic behavior of s^* in Theorem 2 are omitted due to space restrictions.

3 Randomized Search

3.1 Lower Bound for Randomized k-Max-Search

We consider $k = 1$ first. The optimal deterministic online algorithm achieves a competitive ratio of $r^*(1,\varphi) = \sqrt{\varphi}$. As shown in [2], randomization can dramatically improve this. Assume for simplicity, that $\varphi = 2^\ell$ for some integer ℓ. For $0 \leq j < \ell$ let $\mathsf{RPP}(j)$ be the reservation price policy with reservation $m2^j$, and define EXPO to be a uniform probability mixture over $\{\mathsf{RPP}(j)\}_{j=0}^{\ell-1}$.

Lemma 5 (Levin, see [2]). *Algorithm* EXPO *is $\mathcal{O}(\log\varphi)$-competitive.*

We shall prove that EXPO is in fact the optimal randomized online algorithm for 1-max-search. We will use the following version of *Yao's principle* [18].

Theorem 5 (Yao's principle). *For an online maximization problem denote by $\mathcal{S}$ the set of possible input sequences, and by $\mathcal{A}$ the set of deterministic algorithms, and assume that $\mathcal{S}$ and $\mathcal{A}$ are finite. Fix any probability distribution $y(\sigma)$*

on $\mathcal{S}$, and let S be a random sequence according to this distribution. Let RALG *be any mixed strategy, given by a probability distribution on $\mathcal{A}$. Then,*

$$\mathrm{CR}(\mathsf{RALG}) = \max_{\sigma\in\mathcal{S}} \frac{\mathsf{OPT}(\sigma)}{\mathbb{E}[\mathsf{RALG}(\sigma)]} \geq \left(\max_{\mathsf{ALG}\in\mathcal{A}} \mathbb{E}\left[\frac{\mathsf{ALG}(\mathrm{S})}{\mathsf{OPT}(\mathrm{S})}\right]\right)^{-1}. \tag{8}$$

The reader is referred to standard textbooks for a proof (e.g. chapter 6 and 8 in [19]). In words, Yao's principle says that we obtain a lower bound on the competitive ratio of the best randomized algorithm by calculating the performance of the best deterministic algorithm for a chosen probability distribution of input sequences. Note that (8) gives a lower bound for *arbitrary* chosen input distributions. However, only for well-chosen y's we will obtain strong lower bounds.

We first need to establish the following lemma on the representation of an arbitrary randomized algorithm for k-search.

Lemma 6. *Let* RALG *be a randomized algorithm for the k-max-search problem. Then* RALG *can be represented by a probability distribution on the set of all deterministic algorithms for the k-max-search problem.*

The proof is omitted due to space restrictions. The next lemma yields the desired lower bound.

Lemma 7. *Let $\varphi > 1$. Every randomized 1-max-search algorithm* RALG *satisfies*

$$\mathrm{CR}(\mathsf{RALG}) \geq (\ln\varphi)/2.$$

Proof. Let $b > 1$ and $\ell = \log_b \varphi$. We define a finite approximation of $\mathcal{I}$ by $\mathcal{I}_b = \{mb^i \mid i = 0\ldots\ell\}$, and let $\mathcal{P}_b = \bigcup_{n\geq k} \mathcal{I}_b^n$. We consider the 1-max-search problem on $\mathcal{P}_b$. As $\mathcal{P}_b$ is finite, also the set of deterministic algorithms $\mathcal{A}_b$ is finite. For $0 \leq i \leq \ell - 1$, define sequences of length ℓ by

$$\sigma_i = mb^0, \ldots, mb^i, m, \ldots, m. \tag{9}$$

Let $\mathcal{S}_b = \{\sigma_i \mid 0 \leq i \leq \ell - 1\}$ and define the probability distribution y on $\mathcal{P}_b$ by

$$y(\sigma) = \begin{cases} 1/\ell & \text{for } \sigma \in \mathcal{S}_b, \\ 0 & \text{otherwise.} \end{cases}$$

Let $\mathsf{ALG} \in \mathcal{A}_b$. Note that for all $1 \leq i \leq \ell$, the first i prices of the sequences σ_j with $j \geq i-1$ coincide, and ALG cannot distinguish them up to time i. As ALG is deterministic, it follows that if ALG accepts the i-th price in $\sigma_{\ell-1}$, it will accept the i-th price in all σ_j with $j \geq i-1$. Thus, for every ALG, let $0 \leq \chi(\mathsf{ALG}) \leq \ell-1$ be such that ALG accepts the $(\chi(\mathsf{ALG})+1)$-th price, i.e. $mb^{\chi(\mathsf{ALG})}$, in $\sigma_{\ell-1}$. ALG will then earn $m\,b^{\chi(\mathsf{ALG})}$ on all σ_j with $j \geq \chi(\mathsf{ALG})$, and m on all σ_j with $j < \chi(\mathsf{ALG})$. To shorten notation, we write χ instead of $\chi(\mathsf{ALG})$ in the following. Thus, we have

$$\mathbb{E}\left[\frac{\mathsf{ALG}}{\mathsf{OPT}}\right] = \frac{1}{\ell}\left[\sum_{j=0}^{\chi-1} \frac{m}{mb^j} + \sum_{j=\chi}^{\ell-1} \frac{mb^\chi}{mb^j}\right] = \frac{1}{\ell}\left[\frac{1-b^{-\chi}}{1-b^{-1}} + \frac{1-b^{-(\ell-\chi)}}{1-b^{-1}}\right],$$

where the expectation $\mathbb{E}[\cdot]$ is with respect to the probability distribution $y(\sigma)$. If we consider the above term as a function of χ, then it is easily verified that it attains its maximum at $\chi = \ell/2$. Thus,

$$\max_{\mathsf{ALG}\in\mathcal{A}_b} \mathbb{E}\left[\frac{\mathsf{ALG}}{\mathsf{OPT}}\right] \le \frac{1}{\ell}\left(1-\frac{1}{\sqrt{\varphi}}\right)\frac{2b}{b-1} \le \frac{1}{\ln\varphi}\cdot\frac{2b\ln b}{b-1}. \tag{10}$$

Let Υ_b be the set of all randomized algorithms for 1-max-search with possible price sequences $\mathcal{P}_b$. By Lemma 6, each $\mathsf{RALG}_b \in \Upsilon_b$ may be given as a probability distribution on $\mathcal{A}_b$. Since $\mathcal{A}_b$ and $\mathcal{S}_b$ are both finite, we can apply Theorem 5. Thus, for all $b > 1$ and all $\mathsf{RALG}_b \in \Upsilon_b$, we have

$$\mathrm{CR}(\mathsf{RALG}_b) \ge \left(\max_{\mathsf{ALG}\in\mathcal{A}_b} \mathbb{E}\left[\frac{\mathsf{ALG}}{\mathsf{OPT}}\right]\right)^{-1} \ge \ln\varphi\,\frac{b-1}{2b\ln b}.$$

Let Υ be the set of all randomized algorithms for 1-max-search on $\mathcal{P}$. Since for $b \to 1$, we have $\mathcal{A}_b \to \mathcal{A}$, $\Upsilon_b \to \Upsilon$ and $(b-1)/(2b\ln b) \to \frac{1}{2}$, the proof is completed. □

In fact, Lemma 7 can be generalized to arbitrary $k \ge 1$.

Lemma 8. *Let $k \in \mathbb{N}$, $\varphi > 1$. Let RALG be any randomized algorithm for k-max-search. Then, we have*

$$\mathrm{CR}(\mathsf{RALG}) \ge (\ln\varphi)/2.$$

Giving an optimal randomized algorithm for k-max-search is straightforward. For $1 < b < \varphi$ and $\ell = \log_b\varphi$, EXPO_k chooses j uniformly at random from $\{0,\dots,\ell-1\}$, and sets all its k reservation prices to mb^j.

Lemma 9. *Let $k \in \mathbb{N}$. EXPO_k is an asymptotically optimal randomized algorithm for the k-max-search problem with $\mathrm{CR}(\mathsf{EXPO}_k) = \ln\varphi/\ln 2$ as $\varphi \to \infty$.*

The proofs of Lemma 8 and Lemma 9 are omitted due to space restrictions.

3.2 Randomized k-Min-Search

The proof of the lower bound for k-min-search, Theorem 4, uses an analogous version of Yao's principle (see for instance Theorem 8.5 in [19]).

Proof (Theorem 4). We only give the proof for $k = 1$. Let $\mathcal{S} = \{\sigma_1, \sigma_2\}$ with

$$\sigma_1 = m\sqrt{\varphi}, M, \dots, M \quad \text{and} \quad \sigma_2 = m\sqrt{\varphi}, m, M, \dots, M,$$

and let $y(\sigma)$ be the uniform distribution on $\mathcal{S}$. For $i \in \{1,2\}$, let ALG_i be the reservation price policy with reservation prices $p_1^* = m\sqrt{\varphi}$ and $p_2^* = m$, respectively. Obviously, the best deterministic algorithm against the randomized input given by the distribution $y(\sigma)$ must be either ALG_1 or ALG_2. Since

$$\mathbb{E}\left[\frac{\mathsf{ALG}_i}{\mathsf{OPT}}\right] = (1+\sqrt{\varphi})/2, \quad i \in \{1,2\},$$

the desired lower bound follows from the min-cost version of Yao's principle. The proof for general $k \ge 1$ is straightforward by repeating the prices $m\sqrt{\varphi}$ and m in σ_1 and σ_2 in blocks of k times. □

4 Robust Valuation of Lookback Options

In this section, we use the deterministic k-search algorithms from Section 2 to derive upper bounds for the price of lookback options under the assumption of *bounded stock price paths* and non-existence of arbitrage opportunities. We consider a discrete-time model of trading. For simplicity we assume that the interest rate is zero. The price of the stock at time $t \in \{0, 1, \ldots, T\}$ is given by S_t, with S_0 being the price when seller and buyer enter the option contract. At each price quotation S_t, only one unit of stock (or one lot of constant size) can be traded. In reality, certainly the amount of shares tradeable varies, but we can model this by multiple quotations at the same price. We assume that a lookback option is on a fixed number $k \geq 1$ of shares. Recall that the holder of a lookback call has the right to buy shares from the option writer for the price $S_{min} = \min\{S_t \,|\, 0 \leq t \leq T\}$. Neglecting stock price appreciation, upwards and downwards movement is equally probably. Consequently, we assume a symmetric trading range $[\varphi^{-1/2} S_0, \varphi^{1/2} S_0]$ with $\varphi > 1$. We refer to a price path that satisfies $S_t \in [\varphi^{-1/2} S_0, \varphi^{1/2} S_0]$ for all $1 \leq t \leq T$ as a (S_0, φ) price path.

4.1 Upper Bounds for the Price of Lookback Options

Theorem 6. *Assume $(S_t)_{0 \leq t \leq T}$ is a (S_0, φ) stock price path. Let $s^*(k, \varphi)$ be given by (2), and let*

$$V^*_{Call}(k, S_0, \varphi) = k S_0 (s^*(k, \varphi) - 1)/\sqrt{\varphi}. \tag{11}$$

Let V be the option premium paid at time $t = 0$ for a lookback call option on k shares expiring at time T. Suppose we have $V > V^*_{Call}(k, S_0, \varphi)$. Then there exists an arbitrage opportunity for the option writer, i.e., there is a zero-net-investment strategy which yields a profit for all (S_0, φ) stock price paths.

Proof. In the following, let C_t denote the money in the option writer's cash account at time t. At time $t = 0$, the option writer receives V from the option buyer, and we have $C_0 = V$. The option writer then successively buys k shares, applying RPP for k-min-search with reservation prices as given by (7). Let H be the total sum of money spent for purchasing k units of stock. By Lemma 3 we have $H \leq k s^*(k, \varphi) S_{min}$. At time T the option holder executes her option, buying k shares from the option writer for $k S_{min}$ in cash. Thus, after everything has been settled, we have $C_T = V - H + k S_{min} \geq V + k S_{min}(1 - s^*(k, \varphi))$. Because of $S_{min} \geq S_0/\sqrt{\varphi}$ and $V > V^*_{\mathrm{Call}}(k, S_0, \varphi)$, we conclude that $C_T > 0$ for all possible (S_0, φ) stock price paths. Hence, this is indeed a zero net investment profit for the option writer on all (S_0, φ) stock price paths. □

Under the no-arbitrage assumption, we immediately obtain an upper bound for the value of a lookback call option.

Corollary 1. *Under the no-arbitrage assumption, we have $V \leq V^*_{Call}(k, S_0, \varphi)$, with $V^*_{Call}(k, S_0, \varphi)$ as defined in (11).*

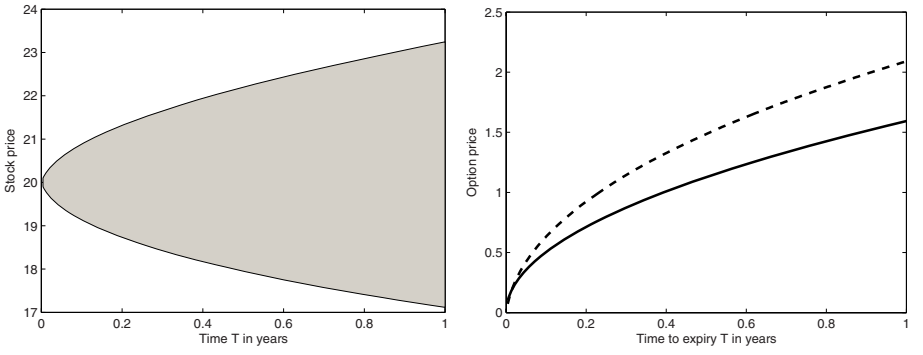


Fig. 1. The left plot shows the expected trading range of a geometric Brownian motion with volatility $\sigma = 0.2$ and $S(0) = 20$. The right plot shows the price of a lookback call with maturity T in the Black-Scholes model (solid line) and the bound V^*_{Call} (dashed line), with $k = 100$ and $\varphi(T)$ chosen to match the expected trading range.

Using Lemma 1 and similar no-arbitrage arguments, also an upper bound for the price of a lookback put option can be established.

Note that it is not possible to derive non-trivial *lower bounds* for lookback options in the bounded stock price model, as a (S_0, φ)-price path may stay at S_0 throughout, making the option mature worthless for the holder. To derive lower bounds, there must be a promised fluctuation of the stock. In the classical Black-Scholes pricing model, this is given by the volatility of the Brownian motion.

4.2 Comparison to Pricing in Black-Scholes Model

Goldman, Sosin and Gatto [20] give closed form solutions for the value of lookback puts and calls in the Black-Scholes setting. Let σ be the volatility of the stock price, modeled by a geometric Brownian motion, $S(t) = S_0 \exp(-\sigma^2 t/2 + \sigma B(t))$, where $B(t)$ is a standard Brownian motion. Let $\Phi(x)$ denote the cumulative distribution function of the standard normal distribution. Then, for zero interest rate, at time $t = 0$ the value of a lookback call on one share of stock, expiring at time T, is given by

$$V^{\text{BS}}_{\text{Call}}(S_0, T, \sigma) = S_0\big(2\,\Phi(\sigma\sqrt{T}/2) - 1\big). \tag{12}$$

The hedging strategy is a certain roll-over replication strategy of a series of European call options. Everytime the stock price hits a new all-time low, the hedger has to "roll-over" her position in the call to one with a new strike. Interestingly, this kind of behavior to act only when a new all-time low is reached resembles the behavior of RPP for k-min-search.

For a numerical comparison of the bound $V^*_{\text{Call}}(k, S_0, \varphi, T)$ with the Black-Scholes-type pricing formula (12), we choose the fluctuation rate $\varphi = \varphi(T)$ such that the expected trading range $\big[\mathbb{E}(\min_{0\le t\le T} S_t), \mathbb{E}(\max_{0\le t\le T} S_t)\big]$ of a

geometric Brownian motion starting at S_0 with volatility σ is $\left[\varphi^{-1/2}S_0, \varphi^{1/2}S_0\right]$. Figure 1 shows the results for $\sigma = 0.2$, $S_0 = 20$ and $k = 10$.

References

1. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Comm. ACM 28(2), 202–208 (1985)
2. El-Yaniv, R., Fiat, A., Karp, R.M., Turpin, G.: Optimal search and one-way trading online algorithms. Algorithmica 30(1), 101–139 (2001)
3. Lippmann, S.A., McCall, J.J.: The economics of job search: a survey. Economic Inquiry XIV, 155–189 (1976)
4. Lippmann, S.A., McCall, J.J.: The economics of uncertainty: selected topics and probabilistic methods. Handbook of mathematical economics 1, 211–284 (1981)
5. Rosenfield, D.B., Shapiro, R.D.: Optimal adaptive price search. Journal of Economic Theory 25(1), 1–20 (1981)
6. Ajtai, M., Megiddo, N., Waarts, O.: Improved algorithms and analysis for secretary problems and generalizations. SIAM Journal on Disc. Math. 14(1), 1–27 (2001)
7. Kleinberg, R.D.: A multiple-choice secretary algorithm with applications to online auctions. In: SODA, pp. 630–631 (2005)
8. Hull, J.C.: Options, Futures, and Other Derivatives. Prentice-Hall, Englewood Cliffs (2002)
9. Black, F., Scholes, M.S.: The pricing of options and corporate liabilities. Journal of Political Economy 81(3), 637–654 (1973)
10. Shreve, S.E.: Stochastic calculus for finance. II. Springer Finance. Springer-Verlag, New York, Continuous-time models (2004)
11. Cont, R.: Empirical properties of asset returns: stylized facts and statistical issues. Quantitative Finance 1, 223–236 (2001)
12. Merton, R.C.: Option pricing when underlying stock returns are discontinuous. Journal of Financial Economics 3(1-2), 125–144 (1976)
13. Cont, R., Tankov, P.: Financial Modelling with Jump Processes. CRC Press, Boca Raton, USA (2004)
14. Heston, S.L.: A closed-form solution for options with stochastic volatility with applications to bond and currency options. Rev. of Fin. Stud. 6(2), 327–343 (1993)
15. DeMarzo, P., Kremer, I., Mansour, Y.: Online trading algorithms and robust option pricing. In: Proc. of the ACM Symp. on Theory of Comp., STOC, pp. 477–486. ACM Press, New York, USA (2006)
16. Epstein, D., Wilmott, P.: A new model for interest rates. International Journal of Theoretical and Applied Finance 1(2), 195–226 (1998)
17. Korn, R.: Worst-case scenario investment for insurers. Insurance Math. Econom. 36(1), 1–11 (2005)
18. Yao, A.C.C.: Probabilistic computations: toward a unified measure of complexity. In: 18th Symp. on Foundations of Comp. Sci., pp. 222–227. IEEE Computer Society Press, Los Alamitos (1977)
19. Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press, New York (1998)
20. Goldman, M.B., Sosin, H.B., Gatto, M.A.: Path dependent options: buy at the low, sell at the high. The Journal of Finance 34(5), 1111–1127 (1979)

Linear Data Structures for Fast Ray-Shooting Amidst Convex Polyhedra*

Haim Kaplan, Natan Rubin, and Micha Sharir

School of Computer Science, Tel Aviv University, Tel Aviv, Israel
{haimk,rubinnat,michas}@post.tau.ac.il

Abstract. We consider the problem of ray shooting in a three-dimensional scene consisting of k (possibly intersecting) convex polyhedra with a total of n facets. That is, we want to preprocess them into a data structure, so that the first intersection point of a query ray and the given polyhedra can be determined quickly. We describe data structures that require $\tilde{O}(n\text{poly}(k))$ preprocessing time and storage, and have polylogarithmic query time, for several special instances of the problem. These include the case when the ray origins are restricted to lie on a fixed line ℓ_0, but the directions of the rays are arbitrary, the more general case when the supporting lines of the rays pass through ℓ_0, and the case of rays orthogonal to z-axis with arbitrary origins. In all cases, this is a significant improvement over previously known techniques (which require $\Omega(n^2)$ storage, even when $k \ll n$).

1 Introduction

The general ray-shooting problem can be defined as follows: *Given a collection Γ of n objects in $\mathbb{R}^d$, preprocess Γ into a data structure so that one can quickly determine the first object in Γ intersected by a query ray.*

The ray-shooting problem has received much attention because of its applications in computer graphics and other geometric problems. Generally, there is a tradeoff between query time and storage (and preprocessing), where faster queries require more storage and preprocessing. In this paper, we focus on the case of fast (polylogarithmic) query time, and seek to minimize the storage of the data structures. For a more comprehensive review of the problem and its solutions, see [12]. If Γ consists of arbitrary triangles in $\mathbb{R}^3$, the best known data structure is due to Pellegrini [11]; it requires $O(n^{4+\epsilon})$ preprocessing and storage and has logarithmic query time. If Γ is the collection of facets on the boundary of a convex polyhedron, then an optimal algorithm, with $O(\log n)$ query and

* Work by Haim Kaplan and Natan Rubin has been supported by Grant 975/06 from the Israel Science Fund. Work by Micha Sharir and Natan Rubin was partially supported by NSF Grant CCF-05-14079, by a grant from the U.S.-Israeli Binational Science Foundation, by grant 155/05 from the Israel Science Fund, Israeli Academy of Sciences, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 287–298, 2007.

$O(n)$ storage, can be obtained using the hierarchical decomposition of Dobkin and Kirkpatrick [9]. For the case where Γ consists of the boundary facets of k convex polyhedra, Agarwal and Sharir [4] describe a solution having polylogarithmic query time and $O(n^{2+\epsilon}k^2)$ storage, which is an improvement over [11]. Still, this solution requires $\Omega(n^2)$ storage, even for small k, in contrast to the $O(n)$ storage achieved in [9] for the case $k = 1$.

A wide range of special instances of the ray-shooting problem have been considered, where restrictions are imposed either on the query rays or on the input polyhedra. The most popular are the cases of C-oriented or fat input polyhedra, as well as the case of vertical ray-shooting (amidst arbitrary convex polyhedra). Another case, studied by Bern *et al.* [6], involves rays whose origins lie on a fixed straight line in $\mathbb{R}^3$. Unfortunately, all the known solutions to the restricted problems mentioned above require $\Omega(n^2)$ storage to achieve polylogarithmic query time, even in the case where the input consists of k convex polyhedra, where k is small relatively to n.

Our contribution. In this paper we focus on the case where the input consists of k convex polyhedra with a total of n facets, and implicitly assume that $k \ll n$. Our goal is to derive algorithms with polylogarithmic query time, for which the storage that they require is (nearly) linear in n. We achieve this in several useful special cases, where one case involves ray-shooting with rays whose origins lie on a fixed line. In this case our solution uses $\tilde{O}(nk^5)$ storage and preprocessing, and answers queries in polylogarithmic time. (The notation $\tilde{O}(\cdot)$ hides polylogarithmic factors.) We extend this solution for the case where the rays lie on lines that pass through a fixed line, but their origins are arbitrary, and for the case of rays orthogonal to the z-axis with arbitrary origins. This is achieved at the cost of increasing the preprocessing time to $\tilde{O}(nk^6)$, with similar time and storage complexity. The problems that we study have the common property that the lines containing the query rays have only three degrees of freedom, as opposed to the general case when the lines have four degrees of freedom. If we ignore the issue of making the performance of the algorithm depend on k, there are solutions that require $O(n^{3+\epsilon})$ storage and preprocessing, with polylogarithmic query time (e.g., a simple adaptation of the technique of [3]). No previous algorithm solves these problems with performance that depends subquadratically on n, when $k \ll n$.

Overview. We apply the following uniform approach to both problem instances. Denote by $\mathcal{P}$ the set of input polyhedra. We define a parametric space $\mathbb{S}$ whose points represent lines containing the query rays. Since the lines containing the query rays have three degrees of freedom, we take $\mathbb{S}$ to be $\mathbb{R}^3$. Each polyhedron $P \in \mathcal{P}$ defines a surface σ_P in $\mathbb{S}$, which is the locus of all (points representing) lines that are tangent to P. The arrangement $\mathcal{C}(\mathcal{P})$ of the surfaces σ_P yields a subdivision of $\mathbb{S}$, each of whose cells C is a maximal connected relatively open set of lines which stab the same subset $\mathcal{P}_C$ of $\mathcal{P}$. When the polyhedra of $\mathcal{P}$ are pairwise disjoint, the order in which lines ℓ in any fixed cell C intersect the polyhedra of $\mathcal{P}_C$ is the same for all $\ell \in C$. Let r be a query ray, let ℓ_r

be the line containing r, and let $C \subset \mathbb{S}$ denote the cell of $\mathcal{C}(\mathcal{P})$ containing the point representing ℓ_r. We can answer a query by a binary search with the ray origin among the sorted list of polyhedra of $\mathcal{P}_C$. In Section 2, we modify the analysis of Brönnimann *et al.* [7] to prove that the complexity of $\mathcal{C}(\mathcal{P})$, for lines ℓ_r passing through a fixed line, is $O(nk^2)$. We use the *persistent* data structure technique of [13] to search for the cell of $\mathcal{C}(\mathcal{P})$ containing ℓ_r. This results in an algorithm for ray-shooting from a line into a collection of pairwise disjoint polyhedra, which requires $\tilde{O}(nk^2)$ storage and preprocessing time, and supports queries in polylogarithmic time.

The above approach fails when the polyhedra of $\mathcal{P}$ may intersect each other, because the order in which the polyhedra of $\mathcal{P}_C$ are intersected by ℓ_r need not be fixed over $\ell_r \in C$. To handle this difficulty, we apply a more sophisticated technique. We add to $\mathcal{P}$ all *pairwise intersections* between the polyhedra of $\mathcal{P}$, and obtain a set $\mathcal{Q}$, consisting of $O(k^2)$ polyhedra with a total of $O(nk)$ facets. Instead of $\mathcal{C}(\mathcal{P})$ we now consider the arrangement $\mathcal{C}(\mathcal{Q})$, which is a refinement of $\mathcal{C}(\mathcal{P})$. Each cell C in $\mathcal{C}(\mathcal{Q})$ is a maximal connected region of $\mathbb{S}$ such that all lines represented in C stab the same subset $\mathcal{Q}_C$ of $\mathcal{Q}$. We can use essentially the same spatial data structure as in the case of disjoint polyhedra, constructed over $\mathcal{C}(\mathcal{Q})$ instead of over $\mathcal{C}(\mathcal{P})$, to locate the cell of $\mathcal{C}(\mathcal{Q})$ containing a query line ℓ_r. Now the structure requires $\tilde{O}(nk \cdot (k^2)^2) = \tilde{O}(nk^5)$ storage and preprocessing, and supports queries in polylogarithmic time; see Section 3.

Let P and Q be any pair of polyhedra in $\mathcal{P}_C$. If $P \cap Q$ does not belong to $\mathcal{Q}_C$, that is, $P \cap Q$ is not intersected by lines represented in C, then P and Q are intersected in the same order by all lines ℓ represented in C. This induces a *partial order* $\prec_C$ on the polyhedra of $\mathcal{P}$. Clearly, $\Xi \subset \mathcal{P}_C$ is an *antichain* with respect to $\prec_C$ if and only if, for all P and Q in Ξ, the polyhedron $P \cap Q$ is in $\mathcal{Q}_C$. The one-dimensional Helly theorem then implies that the polyhedron $\bigcap \Xi$ (the intersection of all of the polyhedra in Ξ) (a) is nonempty, and (b) is intersected by all lines represented in C.

Using *parametric search*, the ray-shooting problem reduces to testing a query segment for intersection with $\mathcal{P}$; see also [2]. The latter problem can be solved, for segments contained in lines represented in C, using a small number of ray-shooting queries amidst polyhedra of the form $\bigcap \Xi$, where Ξ is a *maximal antichain* with respect to $\prec_C$. Organizing the data to facilitate efficient implementation of these ray shootings requires several additional technical steps, which are detailed in the relevant sections below.

The paper is organized as follows. In Section 2 we describe the solution for ray-shooting *from a fixed line* ℓ_0 amidst *disjoint polyhedra*. In Section 3 we describe the extra machinery used to handle *intersecting* polyhedra. In the full version [10], we generalize our approach to solve the ray-shooting problem for rays with *arbitrary origins* that contained in lines which intersect ℓ_0, and for rays orthogonal to the z-axis. For lack of space some proofs are omitted from this extended abstract.

Preliminaries. Let $\mathcal{P}$ be a set of k polyhedra in $\mathbb{R}^3$. A *support vertex* of a line ℓ is a vertex v of a polyhedron $P \in \mathcal{P}$, so that v lies on ℓ (and ℓ does not meet

the interior of P). A *support edge* of a line ℓ is an edge e of a polyhedron $P \in \mathcal{P}$, so that e intersects ℓ at its relative interior (and ℓ does not meet the interior of P). A *support polyhedron* of a line is a polyhedron that contains a support edge or vertex of the line.

Let S be a set of edges and vertices. A line ℓ is a *transversal of* S if ℓ intersect every edge or vertex in S. Line ℓ is an *isolated* transversal of S if it is a transversal of S and it cannot be moved continuously while remaining a transversal of S. A set S of edges and vertices *admits an isolated transversal* if there is an isolated transversal ℓ of S.

Brönnimann *et al.* [7] prove that there are $O(n^2k^2)$ minimal sets of relatively open edges and vertices, chosen from some of the polyhedra, which admit an isolated transversal that is tangent to these support polyhedra. This bound is an easy consequence of the following main lemma of [7].

Theorem 1. *Let P, Q and R be three convex polyhedra in $\mathbb{R}^3$, having p, q and r facets, respectively, and let ℓ_0 be an arbitrary line. There are $O(p+q+r)$ minimal sets of relatively open edges and vertices, chosen from some of these three polyhedra, which, together with ℓ_0, admit an isolated transversal that is tangent to these support polyhedra (excluding sets that lie in planes which contain ℓ_0 and are tangent to all three polyhedra, if such planes exist).*

Clearly, the number of such transversals, over all triples of polyhedra, is $O(nk^2)$. They can be computed in $O(nk^2 \log n)$ time, as described in [7].

2 Ray-Shooting from a Line: Disjoint Polyhedra

In this section we present an algorithm for the case where we shoot from a fixed line ℓ_0, and the input consists of pairwise-disjoint polyhedra. Our approach somewhat resembles that of [7]. We parameterize the set of planes containing ℓ_0 by fixing one such plane Π_0 in an arbitrary manner, and by defining Π_t, for $0 \le t < \pi$, to be the plane obtained by rotating Π_0 around ℓ_0 by angle t.

We represent a line ℓ passing through ℓ_0 by the pair $(t(\ell), D_t(\ell))$, where $t(\ell)$ is such that $\Pi_{t(\ell)}$ contains ℓ, and $D_t(\ell)$ is the point dual to ℓ in $\Pi_{t(\ell)}$. For convenience, we choose $D_t(\cdot)$ to be a planar duality transform which excludes points corresponding to lines parallel to ℓ_0. That is, if we choose in Π_t a coordinate frame in which ℓ_0 is the y-axis and the origin is a fixed point on ℓ_0, then the duality maps each point (ξ, η) to the line $y = \xi x - \eta$ and vice versa. As above, we denote by $\mathbb{S}$ the resulting 3-dimensional parametric space of lines.

We use the arrangement $\mathcal{C}(\mathcal{P})$ defined in the introduction, and recall that each cell C of $\mathcal{C}(\mathcal{P})$ is a maximal connected region in $\mathbb{S}$, such that all lines in C stab the same subset of $\mathcal{P}$. For a cell $C \in \mathcal{C}(\mathcal{P})$, we denote by $\mathcal{P}_C \subseteq \mathcal{P}$ the set of the polyhedra stabbed by the lines in C. Clearly, the polyhedra of $\mathcal{P}_C$ are intersected in the same order by all lines in C (this follows by a simple continuity argument, using the fact that C is connected). This allows us to reduce ray-shooting queries with rays emanating from ℓ_0 to point location queries in $\mathcal{C}(\mathcal{P})$, in the manner explained in the introduction.

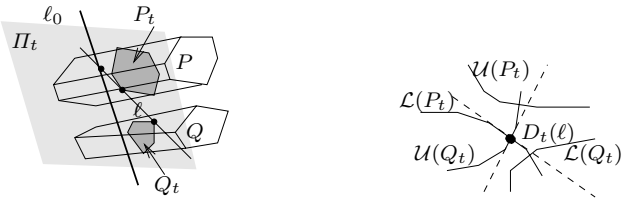


Fig. 1. ℓ is a common tangent of P_t and Q_t in the primal plane Π_t (left). The dual arrangement $\mathcal{A}_t^*$ (right). $D_t(\ell)$ is an intersection of two lines, which are dual to the support vertices of ℓ in Π_t.

Without loss of generality we assume that the interiors of all the polyhedra in $\mathcal{P}$ are disjoint from ℓ_0. Otherwise, we can cut each of the polyhedra, whose interior is intersected by ℓ_0, into two sub-polyhedra, by some plane through ℓ_0. This will not affect the asymptotic complexity of the solution.

This allows us, for each cell C, to order the polyhedra in $\mathcal{P}_C$, and the line ℓ_0, according to their intersections with a line $\ell \in C$, where the order is independent of ℓ. In fact, it suffices to store, for each $C \in \mathcal{C}(\mathcal{P})$, the two polyhedra of $\mathcal{P}_C$ that are adjacent to ℓ_0 in this order. We denote these polyhedra by P_C^+ and P_C^-.

Next we describe how to construct the point location data structure over $\mathcal{C}(\mathcal{P})$, and how to compute P_C^+ and P_C^-, as defined above, for each $C \in \mathcal{C}(\mathcal{P})$.

Point location in $\mathcal{C}(\mathcal{P})$. Consider a plane Π_t, for some fixed value of the rotation angle t. For each polyhedron $P \in \mathcal{P}$, let P_t denote the intersection polygon $P \cap \Pi_t$, and let $\mathcal{P}_t = \{P_t \mid P \in \mathcal{P}, P_t \neq \emptyset\}$ be the set of all resulting (non-empty) polygons. For each polygon $P_t \in \mathcal{P}_t$ we denote by $\mathcal{U}(P_t)$ (resp., $\mathcal{L}(P_t)$) the upper (resp., lower) envelope of all the lines dual to the vertices of P_t in the plane Π_t. The following observation is elementary.

Observation: *Let ℓ be a line contained in Π_t, and let $D_t(\ell)$ be the point dual to ℓ in the dual plane. Then ℓ does not intersect P if and only if $D_t(\ell)$ lies above $\mathcal{U}(P_t)$ or below $\mathcal{L}(P_t)$. Points on $\mathcal{U}(P_t)$ and $\mathcal{L}(P_t)$, are dual to lines tangent to P_t (and, therefore, also to P).* (See Figure 1 for an illustration).

We denote by $\mathcal{A}_t^*$ the arrangement of the monotone piecewise-linear unbounded curves $\{\mathcal{L}(P_t), \mathcal{U}(P_t)\}_{P_t \in \mathcal{P}_t}$. Clearly, regions in $\mathcal{A}_t^*$ correspond to maximal connected sets of lines passing through ℓ_0, such that all lines in the same region stab the same subset of $\mathcal{P}$.[1]

Hence, for each cell $C \in \mathcal{C}(\mathcal{P})$, the intersection of C with the surface $\{t(\ell) = t\}$ consists of one or several connected cells in $\mathcal{A}_t^*$. Thus, answering a point-location query in $\mathcal{C}(\mathcal{P})$, for a query point $(t(\ell), D_t(\ell))$, is reduced to answering a point-location query in $\mathcal{A}_{t(\ell)}^*$, with the point $D_t(\ell)$.

To perform point location queries in $\mathcal{A}_t^*$, for all $t \in [0, \pi]$, we employ an adapted version of the technique of Preparata and Tamassia [13] for point

[1] We regard the polyhedra $P \in \mathcal{P}$ as closed, so tangency of a line to some P also counts as intersection. This requires some (routine) care in handling lower-dimensional faces of $\mathcal{A}_t^*$, which we omit.

location in a dynamically changing monotone planar subdivision. There are certain events, of two different kinds, at which $\mathcal{A}_t^*$ changes combinatorially. Events of the first kind occur when the sweep plane Π_t passes through a vertex v of some $P \in \mathcal{P}$. It can easily be argued that all such events cause $O(nk)$ topology changes to $\mathcal{A}_t^*$ in total (at each of there events, we either get $O(k)$ new vertices of $\mathcal{A}_t^*$, or have to delete $O(k)$ vertices from $\mathcal{A}_t^*$). In events of the second kind, each of the envelopes in $\{\mathcal{L}(P_t), \mathcal{U}(P_t)\}_{P_t \in \mathcal{P}_t}$ is combinatorially unchanged. Events of this kind either involve a concurrent triple of envelopes (where the intersection point is dual to a line tangent to three polygons of $\mathcal{P}_t$), or involve a vertex of one envelope lying on another envelope (which is dual to a line supporting a vertex of one P_t and tangent to another Q_t). That is, each of the listed events corresponds to a set of polyhedra edges and vertices which (together with ℓ_0) admit an isolated transversal. Hence, by Theorem 1, the number of such events is $O(nk^2)$. In summary, we have:

Theorem 2. *Let $\mathcal{P}$ be a set of k convex polyhedra with a total of n facets, let ℓ_0 be a fixed line, and let $\mathcal{C}(\mathcal{P})$ be the corresponding cell decomposition of the parametric space $\mathbb{S}$, as defined above. We can construct, in $O(nk^2 \log^2 n)$ time, a data structure which supports point location queries in $\mathcal{C}(\mathcal{P})$ in $O(\log^2 n)$ time, and requires $O(nk^2 \log^2 n)$ storage.*

Note that this theorem does not require the input polyhedra to be disjoint. We will apply it also in Section 3, in contexts involving intersecting polyhedra.

To compute, for each cell $C \in \mathcal{C}(\mathcal{P})$, the pair of polyhedra P_C^+ and P_C^- "nearest" to ℓ_0, we note that the enumeration of the elements of $\mathcal{P}_C \cup \{\ell_0\}$, ordered according to their intersections with lines $\ell \in C$, changes in at most one position as we pass between neighboring cells of $\mathcal{C}(\mathcal{P})$ (across a common 2-face), where in each such change we have to either insert a new polyhedron P into $\mathcal{P}_C$, or remove a polyhedron from this list. So we maintain this list in a persistent search tree, as follows. After $\mathcal{C}(\mathcal{P})$ has been constructed, we construct a spanning tree of the adjacency graph of its cells, and turn it into a linear-size sequence of cells, each consecutive pair of which are adjacent. We then traverse the sequence, and when we pass from a cell C to the next (adjacent) cell C', we update $\mathcal{P}_C$ in a persistent manner, and construct from it $\mathcal{P}_{C'}$, by the corresponding insertion or deletion of a polyhedron.

Theorem 3. *Let $\mathcal{P}$ be a set of k pairwise disjoint convex polyhedra in $\mathbb{R}^3$ with a total of n facets, and let ℓ_0 be a fixed line. Then we can construct, in $O(nk^2 \log^2 n)$ time, a data structure of size $O(nk^2 \log^2 n)$, which supports ray-shooting queries from ℓ_0 in $O(\log^2 n)$ time.*

3 Ray-Shooting from a Line: Handling Intersecting Polyhedra

The solution described in the previous section fails for the case of intersecting polyhedra because lines which belong to the same cell C may intersect the

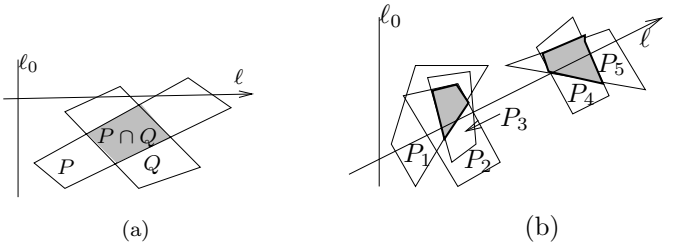


Fig. 2. (a) $P \prec_C Q$, for the cell C containing ℓ; the polyhedron $P \cap Q$ is not in $\mathcal{Q}_C$. (b) Illustrating $\overline{Front}(C)$, for the cell C containing ℓ. We have $Front(C) = \{P_1, P_2\}$ and $\overline{Front}(C) = \{P_1, P_2, P_3\}$. The maximal antichains are $\{P_1, P_2, P_3\}$ and $\{P_4, P_5\}$.

boundaries of the polyhedra of $\mathcal{P}_C$ in different orders. In this section we consider this case, and derive a solution which is less efficient, but still nearly linear in n.

As described in Section 1, we use parametric search to reduce the problem to segment intersection detection. Specifically, the problem is reduced to testing a segment $s = o_1 o_2$, such that o_1 is on ℓ_0, for intersection with the polyhedra of $\mathcal{P}$. We begin with the following trivial observation.

Observation: *Let $\mathcal{P}$ be a set of convex polyhedra, let $\mathcal{A}$ denote the arrangement of their boundaries, and let s be a segment. If the endpoints of s belong to* distinct cells *of $\mathcal{A}$ then s intersects the boundary of a polyhedron of $\mathcal{P}$.*

The first ingredient of our algorithm is thus a data structure for point location in $\mathcal{A}$. For example, we can use the structure described in [13]. For this, we note that $\mathcal{A}$ is not *xy-monotone*, as required in [13]. Nevertheless, we can replace each polyhedron $P \in \mathcal{P}$ by two semi-unbounded polyhedra P^+, P^-, where P^+ (resp., P^-) consists of all the points that lie in P or above (resp., below) it. We obtain $2k$ convex polyhedra, still with a total of $O(n)$ facets, whose arrangement is xy-monotone, and is in fact a refinement of $\mathcal{A}$. Thus, locating a point in the new arrangement, using the algorithm in [13], gives us the cell of $\mathcal{A}$ that contains it. The complexity of the (extended) $\mathcal{A}$ is $O(nk^2)$—an easy and well known fact. Hence, the resulting data structure requires $O(nk^2 \log^2 n)$ storage and preprocessing, and answers point-location queries in $O(\log^2 n)$ time.

The more difficult part is to test whether s intersects a polyhedron of $\mathcal{P}$ when the endpoints of s belong to the same cell of $\mathcal{A}$. In this case, it is sufficient to test s for intersection with the polyhedra of $\mathcal{P}$ which do not contain o_1. Indeed, if s intersects a polyhedron P containing o_1 then the other endpoint o_2 is outside of P, and, therefore, o_1 and o_2 belong to different cells of $\mathcal{A}$.

As in the previous section, we cut each of the polyhedra, whose interior is intersected by ℓ_0, into two sub-polyhedra, by some plane through ℓ_0, and leave rest of the polyhedra unchanged. Note, that we have just added artificial facets to the polyhedra of $\mathcal{P}$, whose affine extensions contain ℓ_0.

We replace $\mathcal{P}$ with the set $\mathcal{Q} = \{P \cap Q \mid P, Q \in \mathcal{P}\}$; note that the original polyhedra of $\mathcal{P}$ are also in $\mathcal{Q}$. The new set contains $O(k^2)$ convex polyhedra of total complexity $O\left(\sum_{P,Q\in\mathcal{P}}(n_P + n_Q)\right) = O(nk)$, where n_P is the number of

facets of polyhedron P. As above, let $\mathcal{C}(\mathcal{Q})$ denote the 3-dimensional arrangement in $\mathbb{S}$ defined by the surfaces of tangents to the polyhedra in $\mathcal{Q}$.

We construct a point location structure over $\mathcal{C}(\mathcal{Q})$ as we did for $\mathcal{C}(\mathcal{P})$ in Section 2. Each cell $C \in \mathcal{C}(\mathcal{Q})$ is a maximal connected region with the property that all lines in C stab the same subset of polyhedra of $\mathcal{Q}$, which we denote by $\mathcal{Q}_C$. This point location data structure supports queries in $O(\log^2(nk)) = O(\log^2 n)$ time, and requires $O(nk \cdot (k^2)^2 \log^2(nk)) = O(nk^5 \log^2 n)$ storage and preprocessing time. The following is an obvious but crucial observation, which justifies the introduction of $\mathcal{Q}$.

Observation: *Let C be a cell of $\mathcal{C}(\mathcal{Q})$, and let ℓ be a line in C. For any pair of distinct polyhedra $P, Q \in \mathcal{P}_C$, the segments $P \cap \ell$ and $Q \cap \ell$ intersect if and only if $P \cap Q$ is in $\mathcal{Q}_C$.*

We consider a query ray in a line ℓ to be *positive* if it is contained in the halfplane of $\Pi_{t(\ell)}$ which lies to the right of ℓ_0 (in an appropriate coordinate frame, as described above). We focus below on the case of positive rays, since negative rays can be handled in a fully symmetric manner. We denote by ℓ^+ the positive ray (emanating from ℓ_0) contained in the line ℓ, and, for a cell $C \in \mathcal{C}(\mathcal{Q})$, we denote by $\mathcal{P}_C^+ \subseteq \mathcal{P}$ the subset of polyhedra of $\mathcal{P}$ intersected by ℓ^+, for any line ℓ in C; since we have assumed that no polyhedron of $\mathcal{P}$ intersects ℓ_0, this property is indeed independent of the choice of ℓ in C.

We define $Front(C)$ to be the set of all polyhedra $P \in \mathcal{P}_C^+$ for which there exists a line ℓ in C, such that an endpoint of $P \cap \ell$ is the closest to ℓ_0, among all the boundary points, along ℓ. That is, $Front(C)$ consists of all candidate polyhedra to be the first intersected by positive rays ℓ^+, for $\ell \in C$.

We define a partial order $\prec_C$ on the polyhedra of $\mathcal{P}_C$, as follows. For $P, Q \in \mathcal{P}_C$, we say that $P \prec_C Q$ if $P \cap Q \notin \mathcal{Q}_C$ (the polyhedron $P \cap Q$ is not stabbed by any line in C), and the segment $P \cap \ell$ appears before $Q \cap \ell$ along ℓ, for any line ℓ in C. See Figure 2 (a) for an illustration.

Lemma 1. *The order $P \prec_C Q$ (a) is well defined and is independent of the choice of ℓ in C, and (b) is indeed a partial order.*

Recall that a subset $\mathcal{T} \subseteq \mathcal{P}_C$ is called an *antichain* of $\prec_C$ if any two distinct polyhedra $P, Q \in \mathcal{T}$ are unrelated under $\prec_C$; $\mathcal{T}$ is called a *maximal antichain* if no (proper) superset of $\mathcal{T}$ is an antichain.

Lemma 2. *A subset $\mathcal{T} \subseteq \mathcal{P}_C$ is an antichain under $\prec_C$ if and only if every line ℓ in C intersects the polyhedron $\bigcap \mathcal{T} = \bigcap_{P \in \mathcal{T}} P$.*

Proof. Let $\mathcal{T}$ be an antichain under $\prec_C$. For any $P, Q \in \mathcal{T}$, the polyhedra P and Q are unrelated under $\prec_C$, so $P \cap Q \in \mathcal{Q}_C$. Therefore, for every line ℓ in C, the segments $P \cap \ell$ and $Q \cap \ell$ intersect. Hence, by the one-dimensional Helly theorem, $\bigcap_{P \in \mathcal{T}} (P \cap \ell) = (\bigcap \mathcal{T}) \cap \ell$ is not empty, for every $\ell \in C$.

To prove the converse statement, let $\mathcal{T}$ be a subset of $\mathcal{P}_C$ such that $\bigcap \mathcal{T}$ is stabbed by every line in C. In particular, for any pair of polyhedra P and Q in $\mathcal{T}$, the polyhedron $P \cap Q$ is stabbed by every line in C. Thus, P and Q are unrelated under $\prec_C$. □

We define $\overline{Front}(C) \subseteq \mathcal{P}_C$ to be the set of all minimal polyhedra under $\prec_C$. It is easy to see that $Front(C) \subseteq \overline{Front}(C)$. (A proper inclusion is possible—see Figure 2 (b).) Since $\overline{Front}(C)$ is a maximal antichain under $\prec_C$, we obtain the following corollary. See Figure 2(b) for an illustration.

Corollary 1. *Every line in C intersects the polyhedron $\bigcap \overline{Front}(C)$.*

For each cell C we store a pointer to a data structure which supports ray-shooting queries at the polyhedron $\bigcap \overline{Front}(C)$ with positive rays contained in lines $\ell \in C$. Details about the structure, and the analysis of its storage and construction cost, will be presented later in this section.

Answering Queries. The query algorithm is based on the following lemma. See Figure 3(a) for an illustration.

Lemma 3. *Let $s = o_1o_2$ be a query segment (where $o_1 \in \ell_0$), contained in a positive ray. Let ℓ be the line containing s, and let C be the cell of $\mathcal{C}(\mathcal{Q})$ that contains ℓ. Then s intersects a (non-artificial) facet on the boundary of some polyhedron of $\mathcal{P}$ if and only if one of the following two conditions holds:*

(i) o_1 and o_2 belong to distinct cells of $\mathcal{A}$.
(ii) s intersects the boundary of the polyhedron $\bigcap \overline{Front}(C)$ and o_2 is not contained in $\bigcap \overline{Front}(C)$.

Proof. If the first one of the two conditions holds then s clearly intersects the boundary of some polyhedron of $\mathcal{P}$. Now assume s intersects the boundary of $\bigcap \overline{Front}(C)$ and o_2 is not contained in $\overline{Front}(C)$. In particular, there is $P \in \overline{Front}(C)$ such that o_2 is not contained in P, and s intersects P. Hence, s intersects a non-artificial facet on the boundary of P.

For the converse statement, assume that s intersects a non-artificial facet on the boundary of some polyhedron P of $\mathcal{P}$. If o_2 is contained in P then, by assumption, o_1 and o_2 belong to distinct cells of $\mathcal{A}$. Otherwise, let P be the first polyhedron intersected by s, and let ℓ^+ denote the ray which contains s. Since o_2 is not contained in P, the segment $P \cap \ell^+$ is contained in s. By definition, P belongs to $\overline{Front}(C)$. By Corollary 1, ℓ^+ intersects $\bigcap \overline{Front}(C)$. Since $\overline{Front}(C) \subset P$, the intersection of ℓ^+ with $\bigcap \overline{Front}(C)$ is also contained in s. Hence, o_2 is not contained in $\bigcap \overline{Front}(C)$, and s intersects $\bigcap \overline{Front}(C)$. □

Now we are ready to describe our query algorithm. First, we use the data structure for point location in $\mathcal{A}$ to test whether the endpoints of s belong to different cells of $\mathcal{A}$. If this is the case, we report intersection and finish. Otherwise, we query the spatial point location structure of $\mathcal{C}(\mathcal{Q})$ to find the cell C which contains the line ℓ containing s. Then, we test whether s intersects $\bigcap \overline{Front}(C)$, and whether o_2 is not contained in $\overline{Front}(C)$, and report intersection if and only if the answer is positive. The correctness of this procedure follows from the preceding analysis.

Query Time and Storage. For any collection of polyhedra $\mathcal{P}' \subseteq \mathcal{P}$, and a face f of $\mathcal{A}$, we say that f is *good* for $\mathcal{P}'$ if $\mathcal{P}'$ is the set of all (closed) polyhedra in $\mathcal{P}$ that contain f.

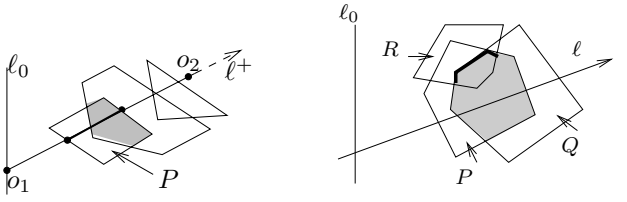


Fig. 3. (a) Query algorithm: case (ii). o_1 and o_2 lie in the same cell of A, s intersects $\bigcap \overline{Front}(C)$ and o_2 lies out of $\bigcap \overline{Front}(C)$. (b) Good facets. For the cell C which contains ℓ, $\bigcap \overline{Front}(C) = P \cap Q$. The facets of $P \cap Q$ which are contained in R (bold) are not good and are not intersected by lines in C.

Lemma 4. *Let C be a cell of $\mathcal{C}(\mathcal{Q})$. If a line ℓ in C intersects $f \in \mathcal{A}$, and f is on $\partial(\bigcap \overline{Front}(C))$, then f is good for $\overline{Front}(C)$.*

Proof. We have to prove that the set of polyhedra that contain f is equal to $\overline{Front}(C)$. Since f is on $\partial(\bigcap \overline{Front}(C))$, it is clear that every polyhedron in $\overline{Front}(C)$ contains f. Conversely, let P be a polyhedron that contains f. We claim that P is minimal in $\mathcal{P}_C^+$ under $\prec_C$, and therefore $P \in \overline{Front}(C)$. Indeed, if P is not minimal under $\prec_C$, then there exists $Q \in \overline{Front}(C)$ such that $Q \prec_C P$. It follows that $Q \cap \ell$ is disjoint from $P \cap \ell$, but this contradicts the fact that ℓ intersects f, which clearly lies in $P \cap Q$. □

Therefore, it suffices to solve, for each cell C, the ray-shooting problem only amidst the faces of $\mathcal{A}$ that lie on the boundary of $\bigcap \overline{Front}(C)$ and are good for $\overline{Front}(C)$. See Figure 3(b) for an illustration.

Let $\mathcal{P}' \subseteq \mathcal{P}$ be a set of polyhedra. Let f be a good facet for $\mathcal{P}'$, let h_f be the plane containing f, and let h_f^+ be the halfspace bounded by h_f and containing $\bigcap \mathcal{P}'$. We define $G(\mathcal{P}')$ to be the polyhedron obtained by intersecting the halfspaces h_f^+, over all good faces f for $\mathcal{P}'$. Note that $\bigcap P' \subseteq G(P')$, and that every good face for P' lies on $\partial G(P')$. In particular, we have the following property (brought without proof).

Lemma 5. *Let C be a cell in $\mathcal{C}(\mathcal{P})$, let ℓ be a line in C, and let ℓ^+ be the positive ray of ℓ, as defined above. Then $\bigcap \overline{Front}(C)$ and $G(\overline{Front}(C))$ have the same intersection with ℓ^+.*

Hence, s intersects $\bigcap \overline{Front}(C)$, and o_2 is not in $\overline{Front}(C)$ if and only if it intersects $G(\overline{Front}(C))$, and o_2 is not in $G(\overline{Front}(C))$. It suffices to test s for the latter property.

For each cell $C \in \mathcal{C}(\mathcal{Q})$, we construct a data structure which supports ray-shooting queries at $G(\overline{Front}(C))$. If implemented as in [9], the structure uses storage linear in the complexity of $G(\overline{Front}(C))$, and can answer queries in $O(\log n)$ time. With each cell $C \in \mathcal{C}(\mathcal{Q})$ we store a pointer to the ray-shooting structure at $G(\overline{Front}(C))$. As each face of $\mathcal{A}$ is good for exactly one subset $\mathcal{P}'$

of polyhedra, the total storage and preprocessing required for all these auxiliary ray-shooting data-structures is $O(nk^2 \log n)$.

Hence, the resulting data structure uses $O(nk^5 \log^2 n)$ storage (the major part of which is consumed by the point location structure in the decomposition $\mathcal{C}(\mathcal{Q})$). Since we use parametric search, the actual query time is quadratic in the cost of a segment intersection query, and is thus $O(\log^4 n)$.

Preprocessing. The only nontrivial part of the preprocessing is the construction of the ray-shooting structures at the polyhedra $G(\overline{Front}(C))$, for all the cells $C \in \mathcal{C}(\mathcal{Q})$. First, for each face f of $\mathcal{A}$, we compute the unique subset $\mathcal{P}_f \subseteq \mathcal{P}$ for which f is good, which is done by testing, for each polyhedron $P \in \mathcal{P}$, whether P contains some fixed point $p_f \in f$. Since the complexity of $\mathcal{A}$ is $O(nk^2)$, this can be done in total time $O(nk^3 \log n)$. We represent each subset $\mathcal{P}_f$ by a bitvector in $\{0,1\}^k$. Next, we collect the faces f having the same set $\mathcal{P}_f$. This can be done by sorting the $O(nk^2)$ k-long bitvectors $\mathcal{P}_f$, in time $O(nk^3 \log n)$. We construct for each of the resulting equivalence classes $\mathcal{T}$ a ray-shooting structure on the polyhedron $G(\mathcal{T})$.

To supply each cell $C \in \mathcal{C}(\mathcal{Q})$ with a pointer to the ray-shooting structure corresponding to $G(\overline{Front}(C))$, we proceed as follows. Recall that $\mathcal{C}(\mathcal{Q})$ is a refinement of $\mathcal{C}(\mathcal{P})$. For a cell $\tilde{C} \in \mathcal{C}(\mathcal{P})$, the cells $C \in \mathcal{C}(\mathcal{Q})$ contained in $\tilde{C}$ form a connected portion of the adjacency structure induced by the arrangement $\mathcal{C}(\mathcal{Q})$. Let C and C' be a pair of adjacent cells in $\mathcal{C}(\mathcal{Q})$ which are contained in the same cell of $\mathcal{C}(\mathcal{P})$. If the set $\mathcal{P}$ of original polyhedra is in general position, there is a unique pair of polyhedra P and Q such that $\{P \cap Q\} = \mathcal{Q}_C \triangle \mathcal{Q}_{C'}$. Therefore, (P, Q) is the only pair of polyhedra related under $\prec_C$ and unrelated under $\prec_{C'}$ (or vice versa). Thus in C we have, say, $P \prec_C Q$, so Q is not in $\overline{Front}(C)$. Since all other pairs do not change their status in $\prec$, the only possible change from $\overline{Front}(C)$ to $\overline{Front}(C')$ is that Q may be added to the latter set, which is the case if and only if the following property holds. Let ℓ be a line on the common boundary between C and C', and let q be the singleton point in $P \cap Q \cap \ell$. Then Q is added to $Front(C')$ if and only if $q \in \bigcap \overline{Front}(C)$; In other words, we have argued that $|\overline{Front}(C) \triangle \overline{Front}(C')| \leq 1$.

Let $\tilde{C}$ be a cell in $\mathcal{C}(\mathcal{P})$. We pick a representative cell $C_0 \in \mathcal{C}(\mathcal{Q})$ contained in $\tilde{C}$, and compute $\overline{Front}(C_0)$ by brute force, as described above. Next we trace the adjacency structure of $\mathcal{C}(\mathcal{Q})$ within $\tilde{C}$ in breadth-first fashion, and find all the cells $C \in \mathcal{C}(\mathcal{Q})$ contained in $\tilde{C}$. For each newly encountered cell C', reached via a previous cell C, we already have, by construction, the pair $P, Q \in \mathcal{P}$ such that $\{P \cap Q\} = \mathcal{Q}_C \triangle \mathcal{Q}_{C'}$. We can now compute $\overline{Front}(C')$ from $\overline{Front}(C)$ as follows. We take a line ℓ on the common boundary of C and C', compute $P \cap \ell$ and $Q \cap \ell$, obtain their common endpoint q, verify that, say, $P \cap \ell$ precedes $Q \cap \ell$ along ℓ, and test whether $q \in \bigcap \overline{Front}(C')$. If so, $\overline{Front}(C')$ is obtained by either adding or removing Q from $\overline{Front}(C)$ (we know which action to take); otherwise $\overline{Front}(C) = \overline{Front}(C')$.

To make the transition from $\overline{Front}(C)$ to $\overline{Front}(C')$ efficient, we precompute a table on the equivalence classes $\mathcal{T}$ that were constructed at the preliminary

stage. For each class $\mathcal{T}$ and each polyhedron P we store a pointer to $\mathcal{T}' = \mathcal{T} \triangle \{P\}$.

To bound the total preprocessing time, we note that the brute force method for finding $\overline{Front}(C)$ takes $O(k \log n)$ time and is applied to $O(nk^2)$ representatives of cells in $\mathcal{C}(\mathcal{P})$. Also observe that in our traversal of $\mathcal{C}(\mathcal{Q})$, over all cells $\tilde{C} \in \mathcal{C}(\mathcal{P})$, we make a total of $O(nk^5)$ transitions, by the upper bound on the overall complexity of $\mathcal{A}(\mathcal{Q})$. Each such transition takes $O(\log n)$ time. In summary, we have:

Theorem 4. *Let $\mathcal{P}$ be a set of k possibly intersecting convex polyhedra with a total of n facets, and a let ℓ_0 be a fixed line in $\mathbb{R}^3$. Then we can construct a data structure supporting ray-shooting queries with rays emanating from ℓ_0, in $O(\log^4 n)$ time. The structure uses $O(nk^5 \log^2 n)$ storage and preprocessing.*

References

1. Aronov, B., de Berg, M., Gray, C.: Ray shooting and intersection searching amidst fat convex polyhedra in 3-space. In: Proc. 22nd Annu. ACM Sympos. Comput. Geom., pp. 88–94. ACM Press, New York (2006)
2. Agarwal, P.K., Matoušek, J.: Ray shooting and parametric search. SIAM J. Comput. 22, 794–806 (1993)
3. Agarwal, P.K., Matoušek, J.: Range searching with semialgebraic sets. Discrete Comput. Geom. 11, 393–418 (1994)
4. Agarwal, P.K., Sharir, M.: Ray shooting amidst convex polyhedra and polyhedral terrains in three dimensions. SIAM J. Comput. 25, 100–116 (1996)
5. Aronov, B., Pellegrini, M., Sharir, M.: On the zone of a surface in a hyperplane arrangement. Discrete Comput. Geom. 9, 177–186 (1993)
6. Bern, M., Dobkin, D.P., Eppstein, D., Grossman, R.: Visibility with a moving point of view. Algorithmica 11, 360–378 (1994)
7. Brönnimann, H., Devillers, O., Dujmovic, V., Everett, H., Glisse, M., Goaoc, X., Lazard, S., Na, H.-S., Whitesides, S.: Lines and free line segments tangent to arbitrary three-dimensional convex polyhedra. SIAM J. Comput. 37, 522–551 (2007)
8. Cole, R., Sharir, M.: Visibility problems for polyhedral terrains. J. Symb. Comput. 7, 11–30 (1989)
9. Dobkin, D.P., Kirkpatrick, D.: A linear algorithm for determining the separation of convex polyhedra. J. Algorithms 6, 391–395 (1985)
10. Kaplan, H., Rubin, N., Sharir, M.: Linear Data Structures for Fast Ray-Shooting amidst Convex Polyhedra `http://www.cs.tau.ac.il/~rubinnat/fastRaySh.pdf`
11. Pellegrini, M.: Ray Shooting on Triangles in 3-Space. Algorithmica 9, 471–494 (1993)
12. Pellegrini, M.: Ray shooting and lines in space. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, 2nd edn., Boca Raton, FL, pp. 839–856. Chapman & Hall/CRC Press (2004)
13. Preparata, F.P., Tamassia, R.: Efficient point location in a convex spatial cell complex. SIAM J. Comput. 21, 267–280 (1992)

Stackelberg Strategies for Atomic Congestion Games

Dimitris Fotakis

Dept. of Information and Communication Systems Engineering,
University of the Aegean, 83200 Samos, Greece
fotakis@aegean.gr

Abstract. We investigate the effectiveness of Stackelberg strategies for atomic congestion games with unsplittable demands. In our setting, only a fraction of the players are selfish, while the rest are willing to follow a predetermined strategy. A *Stackelberg strategy* assigns the coordinated players to appropriately selected strategies trying to minimize the performance degradation due to the selfish players. We consider two orthogonal cases, namely linear congestion games with arbitrary strategies and congestion games on parallel links with arbitrary non-negative and non-decreasing latency functions. We restrict our attention to pure Nash equilibria and derive strong upper and lower bounds on the Price of Anarchy under different Stackelberg strategies.

1 Introduction

Congestion games provide a natural model for non-cooperative resource allocation in large-scale communication networks and have been the subject of intensive research in algorithmic game theory. In a *congestion game* [15], a finite set of non-cooperative players, each controlling an unsplittable unit of load, compete over a finite set of resources. All players using a particular resource experience a cost (or latency) given by a non-negative and non-decreasing function of the resource's load (or congestion). Each player selects her strategy selfishly trying to minimize her *individual cost*, that is the sum of the costs for the resources in her strategy. A natural solution concept is that of a *pure Nash equilibrium*, a configuration where no player can decrease her cost by unilaterally changing her strategy.

At the other end, the network manager cares about the public benefit and aims to minimize the *total cost* incurred by all players. Since a Nash equilibrium does not need to optimize the total cost, one seeks to quantify the inefficiency due to selfish behaviour. The *Price of Anarchy* was introduced in [12] and has become a widely accepted measure of the performance degradation due to the players' selfish behaviour. The (pure) Price of Anarchy is the worst-case ratio of the total cost of a (pure) Nash equilibrium to the optimal total cost. Many recent contributions have provided strong upper and lower bounds on the Price of Anarchy (PoA) for several classes of congestion games, mostly linear congestion games and congestion games on parallel links (see e.g. [14, 7, 2, 3, 1]).

In many cases however, only a fraction of the players are selfish, while the rest are willing to follow a strategy suggested by the network manager (see e.g. [11, 17] for motivation and examples). Korilis *et al.* [11] introduced the notion of *Stackelberg routing* as a theoretical framework for this setting. In Stackelberg routing, a central authority coordinates a fixed fraction of the players and assigns them to appropriately selected

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 299–310, 2007.

strategies trying to minimize the performance degradation due to the selfish behaviour of the remaining players. A *Stackelberg strategy* is an algorithm that determines the strategies of the coordinated players. Given the strategies of the coordinated players, the selfish players lead the system to a configuration where they are at a Nash equilibrium. The goal is to find a Stackelberg strategy of minimum PoA, that is the worst-case ratio of the total cost of all (coordinated and selfish) players in such a configuration to the optimal total cost. Now the PoA is a non-increasing function of the fraction of coordinated players and ideally is given by a continuous curve decreasing from the PoA when all players are selfish to 1 if all players are coordinated (aka a *normal* curve [10]).

In this work, we investigate the effectiveness of Stackelberg routing for atomic congestion games. We consider two essentially orthogonal settings, namely linear congestion games with arbitrary strategies and congestion games on parallel links with arbitrary non-negative and non-decreasing latency functions. We restrict our attention to pure Nash equilibria and obtain strong upper and lower bounds on the pure PoA under different Stackelberg strategies. To the best of our knowledge, this is the first work on Stackelberg routing for atomic congestion games with unsplittable demands.

Related Work. Lücking *et al.* [14] were the first to consider the PoA of atomic congestion games for the objective of total cost[1]. For the special case of uniformly related parallel links, they proved that the PoA is $4/3$. For parallel links with polynomial latency functions of degree d, Gairing *et al.* [7] proved that the PoA is at most $d+1$. Awerbuch *et al.* [2] and Christodoulou and Koutsoupias [3] proved independently that the PoA of congestion games is $5/2$ for linear latencies and $d^{\Theta(d)}$ for polynomial latencies of degree d. Subsequently, Aland *et al.* [1] presented exact bounds on the PoA of congestion games with polynomial latencies. For non-atomic congestion games, where the number of players is infinite and each player controls an infinitesimal amount of load, Roughgarden [16] proved that the PoA is independent of the strategy space and equal to $\rho(\mathcal{D})$, where ρ depends only on the class of latency functions $\mathcal{D}$. Subsequently, Correa *et al.* [4] gave a simple proof of the same bound by introducing $\beta(\mathcal{D}) = 1 - \frac{1}{\rho(\mathcal{D})}$.

To the best of our knowledge, Stackelberg routing has been investigated only in the context of non-atomic games. Focusing on parallel links, Roughgarden [17] proved that it is NP-hard to compute an optimal Stackelberg configuration and investigated the performance of two natural strategies, Scale and Largest Latency First (LLF). Scale uses the optimal configuration scaled by the fraction of coordinated players, denoted α. LLF assigns the coordinated players to the largest cost strategies in the optimal configuration. Roughgarden proved that the PoA of LLF is $1/\alpha$ for arbitrary latencies and $4/(3+\alpha)$ for linear latencies. Kumar and Marathe [13] presented an approximation scheme for the best Stackelberg configuration on parallel links with polynomial latencies.

Some recent papers [19,5,10] extended the results of Roughgarden [17] in several directions. Swamy [19] and independently Correa and Stier-Moses [5] proved that the PoA of LLF is at most $1 + 1/\alpha$ for series-parallel networks with arbitrary latency functions. In addition, Swamy proved that the PoA of LLF is at most $\alpha + (1-\alpha)\rho(\mathcal{D})$

[1] We cite only the most relevant results on the pure PoA of linear congestion games and congestion games on parallel links for the objective of total cost. For a survey on the PoA of congestion games for total and max cost, see e.g. [6,8].

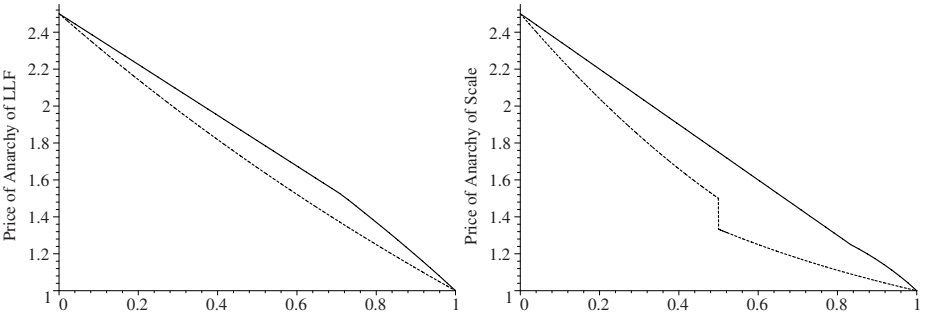


Fig. 1. The upper (solid curves) and lower (dotted curnes) bounds on the PoA of LLF and Scale for linear congestion games as functions of the fraction of coordinated players α. The bounds for LLF are on the left and the bounds for Scale are on the right. The lower bound on the right holds for any optimal-restricted strategy.

for parallel links with latency functions in class $\mathcal{D}$ and obtained upper bounds on the PoA of LLF and Scale for general networks. Karakostas and Kolliopoulos [10] considered non-atomic linear congestion games with arbitrary strategies and presented the best known upper and lower bounds on the PoA of LLF and Scale.

Other recent work on Stackelberg routing for non-atomic games includes [9,18]. In particular, Kaporis and Spirakis [9] showed how to compute efficiently the smallest fraction of coordinated players required to induce an optimal configuration. For the related question of determining the smallest fraction of coordinated players required to improve the cost of a Nash equilibrium, Sharma and Williamson [18] derived a closed expression for parallel links with linear latencies.

Contribution. Motivated by the recent interest in bounding the PoA of LLF and Scale in the non-atomic setting, we investigate the effectiveness of Stackelberg routing in the context of atomic congestion games with unsplittable demands.

For linear congestion games, we derive strong upper and lower bounds on the PoA of LLF and Scale expressed as decreasing functions of the fraction of coordinated players, denoted α (see the plots in Fig. 1). For LLF, we obtain an upper bound of $\min\{(20 - 11\alpha)/8, (3 - 2\alpha + \sqrt{5 - 4\alpha})/2\}$ and a lower bound of $5(2 - \alpha)/(4 + \alpha)$, whose ratio is less than 1.1131. We use a randomized version of Scale, because scaling the optimal configuration by α may be infeasible in the atomic setting. We prove that the expected total cost of the worst configuration induced by Scale is at most $\max\{(5 - 3\alpha)/2, (5 - 4\alpha)/(3 - 2\alpha)\}$ times the optimal total cost. On the negative side, we present a lower bound that holds not only for Scale, but also for any randomized Stackelberg strategy that assigns the coordinated players to their optimal strategies.

An interesting case arises when the number of players is large and the number of coordinated players is considerably larger than the number of resources, even if α is small. To take advantage of this possibility, we introduce a simple Stackelberg strategy called *Cover*. Assuming that the ratio of the number of coordinated players to the number of resources is no less than a positive integer λ, Cover assigns to every resource either at least λ or as many coordinated players as the resource has in the optimal configuration.

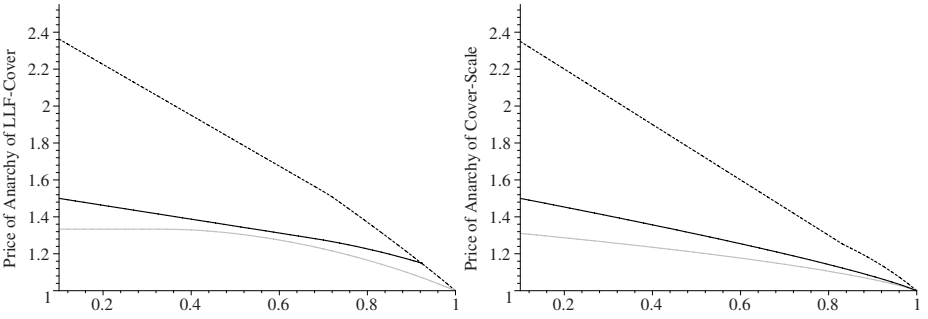


Fig. 2. The PoA of combined LLF-Cover and Cover-Scale vs. the PoA of LLF and Scale for atomic and non-atomic games. Let n be the total number of players, n_s the number of coordinated players, and m the number of resources. For these plots, we let $n = 10m$, assume that $n_s \geq m$, and use 1-Cover. On the x-axis, we have the fraction of coordinated players n_s/n (since $n_s \geq m$, the x-axis starts at 0.1). The solid black curves are the upper bounds on the PoA of atomic linear games under LLF-Cover and Cover-Scale. The dotted black curves are the upper bounds on the PoA of atomic games under LLF and Scale. The solid grey curves are the upper bounds on the PoA of non-atomic linear games under LLF ([10, Theorem 2]) and Scale ([10, Theorem 1]).

We prove that the PoA of Cover tends to the PoA of the corresponding non-atomic linear congestion game as λ grows[2]. More precisely, for linear latencies without constant term, the PoA of Cover is at most $1 + \frac{1}{2\lambda}$ for arbitrary strategies and at most $1 + \frac{1}{4(\lambda+1)^2-1}$ for parallel links. For arbitrary linear latencies, the PoA of Cover is at most $(4\lambda-1)/(3\lambda-1)$. Furthermore, if the ratio of the total number of players n to the number of resources m is large enough (e.g. $n/m \geq 10$), combining Cover with either LLF or Scale gives considerably stronger bounds on the PoA than when using LLF or Scale alone. These bounds are quite close to the best known bounds for non-atomic linear games [10] (see the plots in Fig. 2).

For parallel links, we prove that the PoA of LLF matches that for non-atomic games. In particular, we show that the PoA of LLF is at most $1/\alpha$ for arbitrary latencies and at most $\alpha + (1-\alpha)\rho(\mathcal{D})$ for latency functions in class $\mathcal{D}$.

2 Model, Definitions, and Notation

Congestion Games. A *congestion game* is a tuple $\Gamma(N, E, (\Sigma_i)_{i\in N}, (d_e)_{e\in E})$, where N denotes the set of players, E denotes the set of resources, $\Sigma_i \subseteq 2^E$ denotes the strategy space of each player $i \in N$, and $d_e : \mathbb{N} \mapsto \mathbb{N}$ is a non-negative and non-decreasing latency function associated with each resource $e \in E$. A congestion game is *symmetric* if all players share the same strategy space. A vector $\sigma = (\sigma_1, \ldots, \sigma_n)$ consisting of a strategy $\sigma_i \in \Sigma_i$ for each player $i \in N$ is a *configuration*. For each resource e, let $\sigma_e \equiv |\{i \in N : e \in \sigma_i\}|$ denote the congestion (or load) induced on e by σ. The individual cost of each player i in the configuration σ is $c_i(\sigma) = \sum_{e\in\sigma_i} d_e(\sigma_e)$.

[2] If all players are selfish however, the PoA of a non-symmetric linear congestion game can be 2.5 even if the ratio of the number of players to the number of resources is arbitrarily large.

A configuration σ is a pure *Nash equilibrium* if no player can improve her individual cost by unilaterally changing her strategy. Formally, σ is a pure Nash equilibrium if for every player i and all strategies $s_i \in \Sigma_i$, $c_i(\sigma) \leq c_i(\sigma_{-i}, s_i)$. Rosenthal [15] proved that the pure Nash equilibria of congestion games correspond to the local optima of a natural potential function. Hence every congestion game admits a pure Nash equilibrium.

We mostly consider *linear* congestion games, where every resource e is associated with a linear latency function $d_e(x) = a_e x + b_e$, $a_e, b_e \geq 0$. In the special case of linear functions without constant term (i.e. if $b_e = 0$ for all $e \in E$), we say that the resources are *uniformly related*. We also pay special attention to congestion games on *parallel links*, where the game is symmetric and the common strategy space consists of m singleton strategies, one for each resource.

Social Cost. We evaluate configurations using the objective of *total cost*. The total cost $C(\sigma)$ of a configuration σ is the sum of players' costs in σ. Formally, $C(\sigma) = \sum_{i=1}^{n} c_i(\sigma) = \sum_{e \in E} \sigma_e d_e(\sigma_e)$. An optimal configuration, usually denoted o, minimizes the total cost $C(o)$ among all configurations in $\times_{i \in N} \Sigma_i$. Even though this work is not concerned with the complexity of computing an optimal configuration, we remark that an optimal configuration can be computed in polynomial time for symmetric network congestion games if $xd_e(x)$'s are convex.

Price of Anarchy. The pure *Price of Anarchy* (PoA) is the maximum ratio $C(\sigma)/C(o)$ over all pure Nash equilibria σ. In other words, the PoA is equal to $C(\sigma)/C(o)$, where σ is the pure Nash equilibrium of maximum total cost.

Stackelberg Strategies. We consider a scenario where a *leader* coordinates n_s players, and only $n - n_s$ players are selfish. The leader assigns the *coordinated* players to appropriately selected strategies trying to minimize the performance degradation due to the selfish behaviour of the remaining players. A *Stackelberg configuration* consists of the strategies to which the coordinated players are assigned. A *Stackelberg strategy* is an algorithm computing a Stackelberg configuration[3].

We restrict our attention to *optimal-restricted* Stackelberg strategies that assign the coordinated players to their strategies in the optimal configuration. Given an optimal configuration $o = (o_1, \ldots, o_n)$, an optimal-restricted Stackelberg strategy selects a (possibly random) set $L \subseteq N$, $|L| = n_s$, and assigns the coordinated players to the corresponding strategies in o. For non-symmetric games, L also determines the identities of the coordinated players[4]. The Stackelberg configuration corresponding to L, denoted $s(L)$, is $s(L) = (o_i)_{i \in L}$. By definition, for every (optimal-restricted) Stackelberg strategy and every L, $s_e(L) \leq o_e$ for all $e \in E$. In the following, we let $\alpha = n_s/n$ denote the fraction of players coordinated by the Stackelberg strategy, and let $k = n - n_s$ denote the number of selfish players.

Let $\Gamma(N, E, (\Sigma_i)_{i \in N}, (d_e)_{e \in E})$ be the original congestion game. The congestion game induced by L is $\tilde{\Gamma}_L(N \setminus L, E, (\Sigma_i)_{i \in N \setminus L}, (\tilde{d}_e)_{e \in E})$, where $\tilde{d}_e(x) = d_e(x + s_e(L))$

[3] We highlight the distinction between a Stackelberg strategy, that is an algorithm, and a strategy of some player i, that is an element of Σ_i.

[4] In the terminology of [10], we employ *strong* Stackelberg strategies for non-symmetric games. A strong Stackelberg strategy is free to choose the identities of the coordinated players in addition to their strategies.

for each $e \in E$. The players in $N \setminus L$ are selfish and reach a pure Nash equilibrium of $\tilde{\Gamma}_L$. Since there may be many pure Nash equilibria, we assume that the selfish players reach the *worst* pure Nash equilibrium of $\tilde{\Gamma}_L$, namely the equilibrium $\sigma(L)$ that maximizes the total cost of $\sigma(L) + s(L)$[5]. We let $\sigma(L)$ denote the worst pure Nash equilibrium of $\tilde{\Gamma}_L$ that maximizes $C(\sigma(L) + s(L)) = \sum_{e \in E} (\sigma_e(L) + s_e(L)) d_e(\sigma_e(L) + s_e(L))$. We usually refer to $\sigma(L)$ as the worst Nash equilibrium induced by L (or by $s(L)$). In addition, we let $f(L) = \sigma(L) + s(L)$ denote the worst configuration induced by L (or by $s(L)$)[6]. With this notation in place, the PoA is equal to $C(f(L))/C(o)$.

Largest Latency First (LLF). LLF assigns the coordinated players to the largest cost strategies in o. More precisely, if we index the players in non-decreasing order of their cost in the optimal configuration o, i.e. $c_1(o) \leq \cdots \leq c_n(o)$, the set of coordinated players selected by LLF is $L = \{k+1, \ldots, n\}$.

Scale. We use a randomized version of Scale that selects each different set $L \subseteq N$, $|L| = n_s$, with probability $1/\binom{n}{n_s}$ and adopts the configuration $s(L) = (o_i)_{i \in L}$. Every player $i \in N$ is selected in L with probability $\alpha = n_s/n$ and the expected number of coordinated players on every resource e is αo_e.

Cover. Cover can be applied only if the number of coordinated players is so large that every resource e with $o_e \geq 1$ can be "covered" by at least one coordinated player. Even though this may be achieved with less coordinated players than m, we assume that $n_s \geq m$ for simplicity, and let $\lambda = \lfloor n_s/m \rfloor \geq 1$. Cover selects a set $L \subseteq N$ such that $|L| \leq n_s$ and either $s_e(L) \geq \lambda$ or $s_e(L) = o_e$ for every $e \in E$. Hence $o_e \geq s_e(L) \geq \min\{\lambda, o_e\}$ for all $e \in E$. Given an optimal configuration o, such a set L can be computed efficiently by the greedy λ-covering algorithm. Despite its simplicity, there are instances where Cover outperforms Scale and LLF.

When L is clear from the context, we omit the dependence on L and use s instead of L. In the following, s denotes the Stackelberg configuration, and σ (resp. f) denotes the worst pure Nash equilibrium (resp. configuration) induced by s. We sometimes write α-LLF, (resp. α-Scale, λ-Cover) to denote LLF (resp. Scale, Cover) coordinating at least αn (resp. αn, λm) players.

3 Stackelberg Strategies for Linear Congestion Games

In this section, we establish upper and lower bounds on the PoA of linear congestion games under different Stackelberg strategies. The upper bounds are based on the following lemma.

Lemma 1. *Let o be an optimal configuration, let s be any optimal-restricted Stackelberg configuration, and let f be the worst configuration induced by s. For all $\nu \in (0, 1)$,*

[5] If $s \in \times_{i \in L} \Sigma_i$ is a configuration for $L \subseteq N$ and $\sigma \in \times_{i \in N \setminus L} \Sigma_i$ is a configuration for $N \setminus L$, $f = \sigma + s$ denotes a configuration for N with $f_i = s_i$ if $i \in L$, and $f_i = \sigma_i$ if $i \in N \setminus L$. The notation is motivated by the fact that $f_e = \sigma_e + s_e$ for all $e \in E$.

[6] $\tilde{\Gamma}_L$ may have many different worst pure Nash equilibria. Since we are interested in bounding the PoA, we can assume wlog that for every L, there is a unique worst pure Nash equilibrium $\sigma(L)$ and a unique worst configuration $f(L) = \sigma(L) + s(L)$ induced by L.

(a) $C(f) \leq \sum_{e\in E}[a_e f_e o_e + a_e(o_e - s_e) + b_e o_e]$.
(b) $(1-\nu)C(f) \leq \frac{1}{4\nu}\sum_{e\in E} a_e o_e^2 + \sum_{e\in E} a_e(o_e - s_e) + \sum_{e\in E} b_e o_e - \nu \sum_{e\in E} b_e f_e$.
(c) $(1-\nu)C(f) \leq \frac{1}{4\nu}C(o) + \sum_{e\in E} a_e(o_e - s_e) + (1 - \frac{1}{4\nu})\sum_{e\in E} b_e(o_e - s_e)$.

Proof sketch. To obtain (a), we apply the approach of [3, Theorem 1] and [2, Theorem 3.2] for the selfish players and use the fact that s is optimal-restricted. Then (b) is obtained from (a) by applying the inequality $xy \leq \nu x^2 + \frac{1}{4\nu}y^2$, which holds for all $x, y \in \mathbb{R}$ and all $\nu \in (0,1)$, to the terms $a_e f_e o_e$. Finally, (c) is obtained from (b) using $s_e \leq f_e$ and the inequality $\nu + \frac{1}{4\nu} \geq 1$, which holds for all $\nu \in (0,1)$. □

Largest Latency First. We first prove an upper and a lower bound on the PoA of LLF.

Theorem 1. *The PoA of α-LLF is at most* $\min\{\frac{20-11\alpha}{8}, \frac{3-2\alpha+\sqrt{5-4\alpha}}{2}\}$.

Proof. Starting from Lemma 1a, observing that for all non-negative integers x, y, z with $z \leq y$, $xy + y - z \leq \frac{1}{3}x^2 + \frac{5}{3}y^2 - \frac{11}{12}yz$, and using that $s_e \leq f_e$ and $s_e \leq o_e$ for all $e \in E$, we obtain that

$$C(f) \leq \tfrac{1}{3}C(f) + \tfrac{5}{3}C(o) - \tfrac{11}{12}\textstyle\sum_{e\in E} s_e(a_e o_e + b_e) \leq \tfrac{1}{3}C(f) + \tfrac{20-11\alpha}{12}C(o)$$

For the last inequality, we use that $\sum_{e\in E} s_e(a_e o_e + b_e) = \sum_{i\in L} c_i(o) \geq \alpha C(o)$, because LLF assigns the coordinated players to the largest cost strategies in o. Therefore, the PoA is at most $\frac{20-11\alpha}{8}$.

For the second bound, we start from Lemma 1c and observe that for all $e \in E$, $o_e - s_e \leq o_e(o_e - s_e)$, because o_e and $o_e - s_e$ are non-negative integers. Thus we obtain that for all $\nu \in (0,1)$,

$$(1-\nu)C(f) \leq \tfrac{1}{4\nu}C(o) + \textstyle\sum_{e\in E}(o_e - s_e)(a_e o_e + b_e) \leq (\tfrac{1}{4\nu} + 1 - \alpha)C(o)$$

For the last inequality, we use that $\sum_{e\in E}(o_e - s_e)(a_e o_e + b_e) = C(o) - \sum_{i\in L} c_i(o) \leq (1-\alpha)C(o)$. Therefore, we obtain an upper bound of $\min_{\nu\in(0,1)}(\frac{1}{4\nu} + 1 - \alpha)/(1-\nu)$. Using $\nu = \frac{\sqrt{5-4\alpha}-1}{4(1-\alpha)}$, we conclude that the PoA is at most $\frac{3-2\alpha+\sqrt{5-4\alpha}}{2}$. □

The following theorem gives a lower bound on the PoA of LLF.

Theorem 2. *For every $\alpha \in [0,1)$ and $\varepsilon > 0$, there is a symmetric linear congestion game for which the PoA under α-LLF is at least* $\frac{5(2-\alpha)}{4+\alpha} - \varepsilon$.

Proof. For any fixed $\alpha \in [0,1)$, let n be a positive integer chosen sufficiently large. Wlog we assume that $n_s = \alpha n$ is an integer. Let $k = n - n_s$, and let $L = \{k+1, \ldots, n\}$. We construct a symmetric game with n_s coordinated and k selfish players.

The construction consists of two parts, one for the selfish players and one for the coordinated players. For the selfish players, we employ the instance of [3, Theorem 4]. For the coordinated players, we use n_s singleton parallel strategies. Formally, there are k^2 resources $g_{i,j}$, $i, j \in [k]$, $k^2(k-1)/2$ resources $h_{i,(j,q)}$, $i \in [k]$, $1 \leq j < q \leq k$, and n_s resources r_i, $i \in L$. The latency function of each g-resource is $d_g(x) = x$, the latency function of each h-resource is $d_h(x) = \frac{2}{k+2}x$, and the latency function of each r-resource is $d_r(x) = \frac{(5k-2)k}{2(k+2)}x$.

The g-resources are partitioned in k rows $G_i = \{g_{i,j} : j \in [k]\}$, $i \in [k]$, and in k columns $G^i = \{g_{j,i} : j \in [k]\}$, $i \in [k]$. The h-resources are partitioned in k rows $H_i = \{h_{i,(j,q)} : 1 \leq j < q \leq k\}$, $i \in [k]$. For h-resources, we have k sets of columns $H^i = \{h_{j,(i,q)} : j \in [k], i < q \leq k\} \cup \{h_{j,(q,i)} : j \in [k], 1 \leq q < i\}$, $i \in [k]$. Every H^i contains $k(k-1)$ resources and every resource $h_{i,(j,q)}$ is included in H^j and H^q. The r-resources are partitioned in n_s singleton sets $R_i = \{r_i\}$, $i \in L$. The common strategy space of all players is $\Sigma = \{G_i \cup H_i : i \in [k]\} \cup \{G^i \cup H^i : i \in [k]\} \cup \{R_i : i \in L\}$, i.e. a player can choose either a row strategy $G_i \cup H_i$, or a column strategy $G^i \cup H^i$, or a parallel strategy R_i.

In the optimal configuration o, every player $i \in [k]$ uses the row strategy $G_i \cup H_i$ and every player $i \in L$ uses the parallel strategy R_i. The cost of the optimal configuration is $C(o) = \frac{(5k-2)kn-k^2(k-4)}{2(k+2)}$. Assuming that n is so large that $k > 4$, the cost of the parallel strategies in o is greater than the cost of the row strategies. Hence the configuration of LLF is $s = (R_i)_{i \in L}$. In the worst Nash equilibrium σ induced by s, every selfish player $i \in [k]$ uses the column strategy $G^i \cup H^i$ and the total cost is $C(\sigma + s) = \frac{(5k-2)k(n+k)}{2(k+2)}$. Therefore, the PoA is at least $\frac{(5k-2)(n+k)}{(5k-2)n-k(k-4)}$. Using $k = (1-\alpha)n$, we obtain the lower bound of $5(2-\alpha)/(4+\alpha) - \varepsilon$, where $\varepsilon \leq 3/n$. □

Scale. We proceed to obtain an upper bound on the PoA of Scale. The configuration of Scale s, the worst Nash equilibrium σ induced by s, and the worst configuration f induced by s are random variables uniquely determined by Scale's random choice L. Also for every resource e, the congestion s_e (resp. σ_e, f_e) induced on e by s (resp. σ, f) is a non-negative integral random variable uniquely determined by L.

Let $\Pr[L] = 1/\binom{n}{n_s}$ be the probability that each $L \subseteq N$, $|L| = n_s$, occurs as the choice of Scale. The expected number of coordinated players on every resource e is:

$$\mathbb{E}[s_e] = \sum_{L \subseteq N, |L| = n_s} \Pr[L] s_e(L) = \alpha o_e \,, \tag{1}$$

because every player $i \in N$ is selected in L with probability $\alpha = n_s/n$. The following theorem gives an upper bound on $\mathbb{E}[C(f)] = \sum_{L \subseteq N, |L| = n_s} \Pr[L] C(f(L))$, namely the expected cost of the worst configuration f induced by Scale.

Theorem 3. *Let o be an optimal configuration. The expected cost of the worst configuration f induced by α-Scale is $\mathbb{E}[C(f)] \leq \max\{\frac{5-3\alpha}{2}, \frac{5-4\alpha}{3-2\alpha}\}\, C(o)$.*

Proof sketch. Applying Lemma 1.a for any fixed choice of Scale L, multiplying by $\Pr[L]$, and using linearity of expectation and (1), we obtain that

$$\mathbb{E}[C(f)] \leq \textstyle\sum_{e \in E} [a_e(\mathbb{E}[f_e] + 1 - \alpha) o_e + b_e o_e] \,, \tag{2}$$

where $\mathbb{E}[f_e] = \sum_{L \subseteq N, |L| = n_s} \Pr[L] f_e(L)$ is the expected number of players on resource e in the worst configuration induced by Scale. To complete the proof, we apply the following proposition to the right-hand side of (2).

Proposition 1. *Let y be a non-negative integer, let $\alpha \in [0,1]$, and let X be a non-negative integral random variable. Then,*

$$(\mathbb{E}[X] + 1 - \alpha) y \leq \begin{cases} \frac{1}{3}\,\mathbb{E}[X^2] + (\frac{5}{3} - \alpha) y^2 & \text{for all } \alpha \in [0, \frac{5}{6}] \\ (\alpha - \frac{1}{2})\mathbb{E}[X^2] + (\frac{5}{2} - 2\alpha) y^2 & \text{for all } \alpha \in (\frac{5}{6}, 1] \end{cases}$$

We observe that $(5-3\alpha)/2$ is greater (resp. less) than or equal to $(5-4\alpha)/(3-2\alpha)$ for all $\alpha \in [0,5/6]$ (resp. $\alpha \in [5/6,1]$). First we show that $\mathbb{E}[C(f)] \leq \frac{5-3\alpha}{2} C(o)$, for all $\alpha \in [0,5/6]$. Then we show that $\mathbb{E}[C(f)] \leq \frac{5-4\alpha}{3-2\alpha} C(o)$, for all $\alpha \in (5/6,1]$. □

A Lower Bound for Optimal-Restricted Strategies. The following theorems give a lower bound on the PoA of Scale and any (even randomized) *optimal-restricted* strategy.

Theorem 4. *For every $\alpha \in [0,1)$ and $\varepsilon > 0$, there is a symmetric linear congestion game for which the PoA under any randomized optimal-restricted Stackelberg strategy coordinating a fraction α of the players is at least $\frac{2}{1+\alpha} - \varepsilon$.*

Proof sketch. For any fixed $\alpha \in [0,1)$, we construct a symmetric game with $n_s = \alpha n$ coordinated players and $k = n - n_s$ selfish players. There are $n^2(n-1)/2$ resources $h_{i,(j,q)}$, $i \in [n]$, $1 \leq j < q \leq n$, each with latency function $d_h(x) = x$, and a resource r of constant latency $d_r(x) = (n-1)(\frac{3}{2}n - k) + k - 1$. The common strategy space for all players consists of n row strategies $\{r\} \cup H_i$ and n column strategies H^i, where H_i's and H^i's are defined as in the proof of Theorem 2.

The optimal configuration o assigns every player i to the corresponding row strategy $\{r\} \cup H_i$. The total cost is $C(o) = n(n-1)(2n-k) + O(n^2)$. By symmetry, we can assume that any optimal-restricted Stackelberg strategy selects $L = \{k+1,\ldots,n\}$. In the worst Nash equilibrium σ induced by $s = (\{r\} \cup H_i)_{i \in L}$, every selfish player $i \in [k]$ uses the column strategy H^i and the total cost is $C(\sigma + s) \geq 2n^2(n-1)$. □

Theorem 5. *For every $\alpha \in [0,\frac{1}{2})$ and $\varepsilon > 0$, there is a symmetric linear congestion game for which the PoA under any randomized optimal-restricted Stackelberg strategy coordinating a fraction α of the players is at least $\frac{5-5\alpha+2\alpha^2}{2} - \varepsilon$.*

Proof sketch. Instead of the resource r in the proof of Theorem 4, we use a grid of g-resources as in the proof of Theorem 2, which yields the same strategy space as in [3, Theorem 4]. The latency function of each h-resource is $d_h(x) = x$ and the latency function of each g-resource is $d_g(x) = \gamma x$, where $\gamma = \frac{(n-1)(3n/2-k)+k-1}{2k-n-1}$. The rest of the proof is similar to the proof of Theorem 4. □

Cover. The PoA of Cover tends to the PoA of non-atomic linear congestion games as the ratio of the number of coordinated players to the number of resources grows.

Theorem 6. *If $n_s \geq m$, the PoA of λ-Cover is at most $\frac{4\lambda-1}{3\lambda-1}$, where $\lambda = \lfloor n_s/m \rfloor$. For uniformly related resources, the PoA of λ-Cover is at most $1 + \frac{1}{2\lambda}$.*

Proof. Let s be the configuration of λ-Cover, and let f be the worst configuration induced by s. Since either $s_e = o_e$ or $s_e \geq \lambda$, it holds that $o_e - s_e \leq \frac{1}{4\lambda} o_e^2$ for every resource e. Applying this inequality to Lemma 1c, we obtain that for all $\nu \in (0,1)$,

$$(1-\nu)C(f) \leq \tfrac{1}{4\nu} C(o) + \max\{\tfrac{1}{4\lambda}, 1 - \tfrac{1}{4\nu}\} C(o) = \max\{\tfrac{1}{4\lambda} + \tfrac{1}{4\nu}, 1\} C(o) \qquad (3)$$

Using $\nu = \frac{\lambda}{4\lambda-1}$, we conclude that the PoA of λ-Cover is at most $\frac{4\lambda-1}{3\lambda-1}$.

For uniformly related resources, (3) becomes $(1-\nu)C(f) \leq (\frac{1}{4\nu} + \frac{1}{4\lambda})C(o)$. Using $\nu = \sqrt{\lambda^2+\lambda} - \lambda$, we conclude that the PoA is at most $\frac{2\lambda+2\sqrt{\lambda^2+\lambda}+1}{4\lambda} \leq 1 + \frac{1}{2\lambda}$. □

A careful analysis gives a stronger upper bound for congestion games on uniformly related parallel links. The proof of the following lemma is omitted due to lack of space.

Lemma 2. *If $n_s \geq m$, the PoA of λ-Cover for congestion games on uniformly related parallel links is at most $1 + \frac{1}{4(\lambda+1)^2-1}$, where $\lambda = \lfloor n_s/m \rfloor$.*

Combining Cover with LLF and Scale. If the ratio of the number of players n to the number of resources m is large enough (e.g. $n/m \geq 10$), combining Cover with either LLF or Scale gives considerably stronger upper bounds on the PoA than when using LLF or Scale alone. Throughout this section, we assume that $n_s \geq m$ and let λ be any positive integer not exceeding $\lfloor n_s/m \rfloor$. Also the definition of α is different and does not take the players coordinated by Cover into account.

Combining Cover with LLF. First LLF assigns $n_s - \lambda m$ coordinated players to the largest cost strategies in the optimal configuration o. Let $L^L \subseteq N$, $|L^L| = n_s - \lambda m$, be the set of players assigned by LLF. Then Cover assigns the remaining λm coordinated players wrt $(o_i)_{i \in N \setminus L^L}$. Let $L^C \subseteq N \setminus L^L$, $|L^C| \leq \lambda m$, be the set of players assigned by Cover such that $\min\{\lambda, o_e - s_e(L^L)\} \leq s_e(L^C) \leq o_e - s_e(L^L)$ for all $e \in E$. The joint Stackelberg configuration is $s = (o_i)_{i \in L^L \cup L^C}$.

Theorem 7. *If $n_s \geq m$, let λ be any positive integer not exceeding $\lfloor n_s/m \rfloor$, and let $\alpha = \frac{n_s - \lambda m}{n}$. The PoA of combined α-LLF and λ-Cover is at most*

$$\begin{cases} \frac{4\lambda-1}{3\lambda-1}\left(1 - \frac{\alpha}{4\lambda}\right) & \text{for all } \alpha \in [0, \frac{4\lambda^2}{12\lambda^2-6\lambda+1}] \\ \frac{2-\alpha+\sqrt{4\alpha-3\alpha^2}}{2} & \text{for all } \alpha \in [\frac{4\lambda^2}{12\lambda^2-6\lambda+1}, 1] \end{cases}$$

Proof sketch. Combining the proofs of Theorem 6 and Theorem 1, we obtain that the PoA is at most $\min_{\nu \in (0,1)} [\frac{1}{4\nu} + \max\{\frac{1}{4\lambda}, 1 - \frac{1}{4\nu}\}(1-\alpha)]/(1-\nu)$. The theorem follows by choosing ν appropriately. □

Remark 1. In Theorem 7, $\alpha \leq 1 - \lambda\frac{m}{n}$ and the upper bound remains greater than 1 even if $n_s = n$. To obtain a normal curve, we replace combined LLF-Cover with LLF as soon as n_s/n is so large that the bound of Theorem 1 becomes stronger than the bound of Theorem 7 (see also the left plot of Fig. 2).

Combining Cover with Scale. First Cover assigns (at most) λm coordinated players. Let $L^C \subseteq N$, $|L^C| \leq \lambda m$, be the set of players assigned by Cover such that for all $e \in E$, $\min\{\lambda, o_e\} \leq s_e(L^C) \leq o_e$. Then Scale selects a random set $L^S \subseteq N \setminus L^C$, $|L^S| = n_s - \lambda m$, and assigns the remaining $n_s - \lambda m$ coordinated players to the corresponding optimal strategies. The joint Stackelberg configuration is $s(L^C \cup L^S) = (o_i)_{i \in L^C \cup L^S}$. Hence the joint Stackelberg configuration s and the worst configuration f induced by s are random variables uniquely determined by $L^C \cup L^S$.

For the analysis, we fix an arbitrary choice L^C of Cover. Let $\mathbb{P}\mathrm{r}[L^S] = 1/\binom{n-\lambda m}{n_s-\lambda m}$ be the probability that each set $L^S \subseteq N \setminus L^C$, $|L^S| = n_s - \lambda m$, occurs as the choice of Scale. The following theorem establishes an upper bound on the expected cost $\mathbb{E}[C(f)]$ of the worst configuration f induced by the combined Cover-Scale strategy s, where $\mathbb{E}[C(f)]$ is given by

$$\mathbb{E}[C(f)] = \sum_{L^S \subseteq N \setminus L^C, |L^S| = n_s - \lambda m} \mathbb{P}\mathrm{r}[L^S] C(f(L^C \cup L^S))$$

Theorem 8. *If $n_s \geq m$, let λ be any positive integer not exceeding $\lfloor n_s/m \rfloor$, let $\alpha = \frac{n_s - \lambda m}{n - \lambda m}$, and let o be an optimal configuration. The expected cost of the worst configuration f induced by combining λ-Cover and α-Scale is bounded as follows:*

$$\mathbb{E}[C(f)] \leq \frac{2\lambda(3\alpha-2)+1-\alpha^2-(1-\alpha)\sqrt{\alpha^2-2\alpha(8\lambda^2-4\lambda+1)+16\lambda^2-8\lambda+1}}{2\lambda(4\alpha-3)+2(1-\alpha)} C(o)$$

Proof sketch. Combining the proofs of Theorem 3 and Theorem 6, we show that

$$\mathbb{E}[C(f)] \leq \min_{\nu \in (0,1)} [\max\{\tfrac{1}{4\nu} + \tfrac{1-\alpha}{4\lambda}, 1 - \alpha\nu\}/(1-\nu)]\, C(o)$$

The theorem follows by choosing ν appropriately. □

4 Largest Latency First for Congestion Games on Parallel Links

LLF becomes particularly simple when restricted to parallel links (see also [17, Section 3.2]). LLF indexes the links in non-decreasing order of their latencies in the optimal configuration o, i.e. $d_1(o_1) \leq \cdots \leq d_m(o_m)$, and finds the largest index q with $\sum_{\ell=q}^{m} o_\ell \geq n_s$. In the configuration of LLF, $s_\ell = 0$ for all $\ell < q$, $s_\ell = o_\ell$ for all $\ell > q$, and $s_q = n_s - \sum_{\ell=q+1}^{m}$. Hence q is the first link to which some coordinated players are assigned and $\Lambda = d_q(o_q)$ is a lower bound on the cost of the coordinated players in o.

Theorem 9. *The PoA of LLF for atomic congestion games on parallel links is at most $1/\alpha$, where α is the fraction of players coordinated by LLF.*

Proof sketch. The proof is similar to that of [19, Theorem 3.4]. Let o be the optimal configuration, let s be the configuration of LLF, let σ be the worst Nash equilibrium induced by s, and let $f = \sigma + s$ be the worst configuration induced by s.

Every coordinated player has cost at least Λ in o and $C(o) \geq n_s \Lambda$. The crucial observation is that for every link ℓ with $\sigma_\ell > 0$, $d_\ell(f_\ell) \leq \Lambda$. Hence the total cost of selfish players in f is at most $(n - n_s)\Lambda$, and for every link ℓ with $s_\ell > 0$, $d_\ell(f_\ell) \leq d_\ell(o_\ell)$. The latter holds because either $\sigma_\ell = 0$ and $f_\ell = s_\ell \leq o_\ell$, or both $\sigma_\ell > 0$ and $s_\ell > 0$, in which case $d_\ell(f_\ell) \leq \Lambda \leq d_\ell(o_\ell)$. Therefore, the total cost of coordinated players in f is at most $C(o)$ and $C(f) \leq C(o) + (n - n_s)\Lambda \leq (n/n_s)C(o)$. □

Let $\mathcal{D}$ be a non-empty class of non-negative and non-decreasing latency functions. In [16,4], it is shown that the PoA of non-atomic congestion games with latency functions in class $\mathcal{D}$ is at most $\rho(\mathcal{D}) = \sup_{d \in \mathcal{D}} \sup_{x \geq y \geq 0} \frac{xd(x)}{yd(y)+(x-y)d(x)}$. The following theorem establishes the same upper bound on the PoA of atomic congestion games on parallel links. The proof is omitted due to lack of space.

Theorem 10. *The PoA of atomic congestion games on parallel links with latency functions in class $\mathcal{D}$ is at most $\rho(\mathcal{D})$.*

The next theorem follows easily from Theorem 10 and Theorem 9. The proof is similar to that of [19, Theorem 3.5].

Theorem 11. *For atomic congestion games on parallel links, the PoA of LLF is at most $\alpha + (1-\alpha)\rho(\mathcal{D})$, where α is the fraction of players coordinated by LLF and $\mathcal{D}$ is a class of latency functions containing $\{d_\ell(x + s_\ell)\}_{\ell \in [m]}$.*

Remark 2. For linear latencies, Theorem 11 gives an upper bound of $(4-\alpha)/3$ on the PoA of LLF for atomic congestion games on parallel links, which is quite close to the tight bound of $4/(3+\alpha)$ for the corresponding class of non-atomic games [17].

References

1. Aland, S., Dumrauf, D., Gairing, M., Monien, B., Schoppmann, F.: Exact Price of Anarchy for Polynomial Congestion Games. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 218–229. Springer, Heidelberg (2006)
2. Awerbuch, B., Azar, Y., Epstein, A.: The Price of Routing Unsplittable Flow. In: STOC '05, pp. 57–66 (2005)
3. Christodoulou, G., Koutsoupias, E.: The Price of Anarchy of Finite Congestion Games. In: STOC'05, pp. 67–73 (2005)
4. Correa, J.R., Schulz, A.S., Stier-Moses, N.E.: Selfish Routing in Capacitated Networks. Mathematics of Operations Research 29(4), 961–976 (2004)
5. Correa, J.R., Stier-Moses, N.E.: Stackelberg Routing in Atomic Network Games. Technical Report DRO-2007-03, Columbia Bussiness School (2007)
6. Czumaj, A.: Selfish Routing on the Internet. In: Leung, J. (ed.) Handbook of Scheduling: Algorithms, Models, and Performance Analysis, CRC Press, Boca Raton, USA (2004)
7. Gairing, M., Lücking, T., Mavronicolas, M., Monien, B., Rode, M.: Nash Equilibria in Discrete Routing Games with Convex Latency Functions. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 645–657. Springer, Heidelberg (2004)
8. Gairing, M., Lücking, T., Monien, B., Tiemann, K.: Nash Equilibria, the Price of Anarchy and the Fully Mixed Nash Equilibrium Conjecture. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 51–65. Springer, Heidelberg (2005)
9. Kaporis, A.C., Spirakis, P.G.: The Price of Optimum in Stackelberg Games on Arbitrary Single Commodity Networks and Latency Functions. In: SPAA '06, pp. 19–28 (2006)
10. Karakostas, G., Kolliopoulos, S.: Stackelberg Strategies for Selfish Routing in General Multicommodity Networks. Technical Report AdvOL 2006/08, McMaster University (2006)
11. Korilis, Y.A., Lazar, A.A., Orda, A.: Achieving Network Optima Using Stackelberg Routing Strategies. IEEE/ACM Transactions on Networking 5(1), 161–173 (1997)
12. Koutsoupias, E., Papadimitriou, C.: Worst-case Equilibria. In: Meinel, C., Tison, S. (eds.) STACS 99. LNCS, vol. 1563, pp. 404–413. Springer, Heidelberg (1999)
13. Kumar, V.S.A., Marathe, M.V.: Improved Results for Stackelberg Strategies. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 776–787. Springer, Heidelberg (2002)
14. Lücking, T., Mavronicolas, M., Monien, B., Rode, M.: A New Model for Selfish Routing. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 547–558. Springer, Heidelberg (2004)
15. Rosenthal, R.W.: A Class of Games Possessing Pure-Strategy Nash Equilibria. International Journal of Game Theory 2, 65–67 (1973)
16. Roughgarden, T.: The Price of Anarchy is Independent of the Network Topology. Journal of Computer and System Sciences 67(2), 341–364 (2003)
17. Roughgarden, T.: Stackelberg Scheduling Strategies. SIAM Journal on Computing 33(2), 332–350 (2004)
18. Sharma, Y., Williamson, D.P.: Stackelberg Thresholds in Network Routing Games. In: EC '07 (to appear 2007)
19. Swamy, C.: The Effectiveness of Stackelberg Strategies and Tolls for Network Congestion Games. In: SODA '07 (2007)

Good Quality Virtual Realization of Unit Ball Graphs

Sriram V. Pemmaraju and Imran A. Pirwani

Department of Computer Science,
University of Iowa, Iowa City, IA 52242-1419, USA
{sriram,pirwani}@cs.uiowa.edu

Abstract. The *quality* of an embedding $\Phi : V \mapsto \mathbb{R}^2$ of a graph $G = (V, E)$ into the Euclidean plane is the ratio of $\max_{\{u,v\}\in E} ||\Phi(u) - \Phi(v)||_2$ to $\min_{\{u,v\}\notin E} ||\Phi(u) - \Phi(v)||_2$. Given a unit disk graph $G = (V, E)$, we seek algorithms to compute an embedding $\Phi : V \mapsto \mathbb{R}^2$ of best (smallest) quality. This paper presents a simple, *combinatorial* algorithm for computing a $O(\log^{2.5} n)$-quality 2-dimensional embedding of a given unit disk graph. Note that G comes with no associated geometric information. If the embedding is allowed to reside in higher dimensional space, we obtain improved results: a quality-2 embedding in $\mathbb{R}^{O(1)}$. Our results extend to unit ball graphs (UBGs) in fixed dimensional Euclidean space. Constructing a "growth-restricted approximation" of the given unit disk graph lies at the core of our algorithm. This approach allows us to bypass the standard and costly technique of solving a linear program with exponentially many "spreading constraints." As a side effect of our construction, we get a constant-factor approximation to the minimum clique cover problem on UBGs, described without geometry. Our problem is a version of the well known *localization* problem in wireless networks.

1 Introduction

A graph $G = (V, E)$ is a *d-dimensional unit ball graph (UBG)* if there is an embedding $\Phi : V \mapsto \mathbb{R}^d$ such that $\{u, v\} \in E$ iff $\|\Phi(u) - \Phi(v)\|_2 \leq 1$. Such an embedding Φ of G is called a *realization* of G. When $d = 2$, G is called a *unit disk graph (UDG)*. In this paper, we are interested in the problem of finding a realization Φ of a given UBG. Recently, Aspnes et al. [1] have shown that the problem of computing a realization of a given UDG is NP-hard even if all edge lengths between pairs of neighboring vertices are known. The problem remains NP-hard when all angles between adjacent edges are known [3] and also when all angles plus slightly noisy, pairwise distances are known [2]. Given this situation, we consider the problem of computing an "approximate" realization of the given UBG. Let $G = (V, E)$ be a d-dimensional UBG. Let $\Phi : V \mapsto \mathbb{R}^2$ be an embedding of G into $\mathbb{R}^2$. If G is not a clique, then the *quality* of the embedding Φ is defined as:

$$\frac{\max_{\{u,v\}\in E} \|\Phi(u) - \Phi(v)\|_2}{\min_{\{u',v'\}\notin E} \|\Phi(u') - \Phi(v')\|_2}.$$

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 311–322, 2007.

In case G is a clique, then the *quality* of Φ is simply $\max_{\{u,v\}\in E} \|\Phi(u) - \Phi(v)\|_2$. The specific optimization problem we consider is the following: Given a d-dimensional UBG $G = (V, E)$, for fixed d, find an embedding $\Phi : V \mapsto \mathbb{R}^2$ with best (smallest) quality. We call this the *best quality embedding* problem. This paper focuses on devising an approximation algorithm for the best quality embedding problem.

Motivated by applications in VLSI design, Vempala considers a similar problem [13]: given an undirected graph G, compute a one-to-one assignment of vertices of G to points of a 2-dimensional grid (with unit square cells) such that the maximum Euclidean length of an edge is minimized. Note that G is an arbitrary graph, not just a UBG. Using the random projection approach, Vempala obtains an embedding with maximum length $O(\log^3 n\sqrt{\log\log n})$[1]. Note that the embedding constructed by Vempala's algorithm has quality $O(\log^3 n\sqrt{\log\log n})$ because every pair of vertices are separated by at least one unit distance. Motivated by the *localization* problem in wireless sensor networks, Moscibroda et al. define quality as a measure the "goodness" of a realization [10]. This paper [10] claims an $O(\log^{2.5} n\cdot\sqrt{\log\log n})$-quality embedding for UDGs into $\mathbb{R}^2$, but a subsequent full version of the paper [7] corrects this bound to $O(\log^{3.5} n\cdot\sqrt{\log\log n})$.

1.1 Results and Techniques

In this paper, we present a combinatorial algorithm for computing an $O(\log^{2.5} n)$-quality embedding of any d-dimensional UBG (for any fixed d) into $\mathbb{R}^2$. Our result can be seen as improving Vempala's bound [13] when the input is restricted to be a d-dimensional UBG. However, the most important aspect of our algorithm is that it is combinatorial. Our algorithm avoids the costly first step of Vempala's algorithm in which an exponentially large linear program (LP), that imposes *spreading constraints* on the vertices, is solved via the ellipsoid method. We avoid the spreading constraints approach via a combinatorial algorithm for computing a "growth-restricted approximation" of the given UBG. In a related result, we point out that using the "growth-restricted approximation" of the UBG in combination with a recent result due to Krauthgamer and Lee [6] leads to a quality-2 embedding of any d-dimensional UBG into $\mathbb{R}^{O(d\log d)}$. Since we are primarily interested in small, fixed values of d, this is a quality-2 embedding into constant dimensional space. Our algorithm has three main steps.

1. Constructing a growth-restricted approximation H of G.
2. Constructing a volume respecting embedding of H.
3. Performing a random projecting of the volume respecting embedding in $\mathbb{R}^2$.

We describe Steps (1)-(2) in detail; Step (3) and its analysis are essentially the same as in Vempala's paper [13].

[1] Due to a small error in Lemma 2 in Vempala's paper [13], the bound proved in the paper is $O(\log^4 n)$. However, by using a stronger version of Theorem 2 in his paper, one can obtain an $O(\log^3 n\sqrt{\log\log n})$ bound.

Constructing a Growth-Restricted Approximation. For any graph $G = (V, E)$ and for any pair of vertices $u, v \in V$, let $d_G(u, v)$ denote the shortest path distance between u and v. Let $B_G(v, r) = \{u \in V \mid d_G(u, v) \leq r\}$[2] denote the closed ball of radius r centered at v in G. Define the *growth rate* of G to be

$$\rho_G = \inf\{\rho : |B_G(v, r)| \leq r^\rho \text{ for all } v \in V \text{ and all integers } r > 1\}.$$

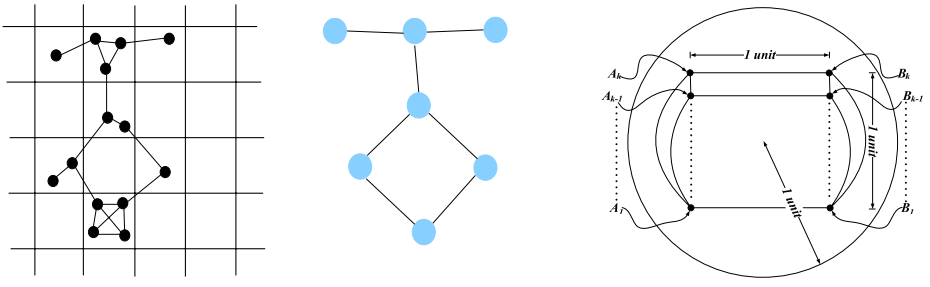
A class $\mathcal{G}$ of graphs is *growth-restricted* if there is some constant c such that for every graph G in $\mathcal{G}$, $\rho_G \leq c$. For any partition $\mathcal{C} = \{C_1, C_2, \ldots\}$ of the vertex set V of G, the *cluster graph of* G *induced by* $\mathcal{C}$, denoted $G[\mathcal{C}]$, is obtained from G by contracting each C_i into a vertex. In Section 2, we show how to partition a given d-dimensional UBG $G = (V, E)$ into cliques $\mathcal{C} = \{C_1, C_2, \ldots\}$ such that the cluster graph $G[\mathcal{C}]$ induced by the clique partition has constant growth rate. Note that if we can construct an α-quality embedding Φ of $G[\mathcal{C}]$ into $\mathbb{R}^L$, we can immediately get an α-quality embedding Φ' of G into $\mathbb{R}^L$: for each vertex v of G set $\Phi'(v) := \Phi(C_i)$, where C_i is the clique that contains v. This allows us to focus on the problem of obtaining a good quality embedding of growth-restricted graphs.

It is quite easy to obtain a clique-partition $\mathcal{C} = \{C_1, C_2, \ldots\}$ with the desired properties if a realization of G is given. For example, given a UDG with 2-dimensional coordinates of vertices known, one can place an infinite grid of $\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}}$ square cells on the plane and obtain a vertex partition $\mathcal{C} = \{C_1, C_2, \ldots\}$ in which each part C_i is all vertices in a cell. Due to the size of the cells each C_i is a clique and furthermore a simple geometric argument shows that there are $O(r^2)$ cells in the radius-$2r$ ball centered at the center of any cell. This suffices to show that in $H := G[\mathcal{C}]$, $|B_H(v, r)| = O(r^2)$, implying that ρ_H is bounded by a constant. See Figure 1(a) for an illustration.

In the absence of geometric information, it is not immediately clear how to come up with an appropriate clique partition. One possible approach is to start with a maximal independent set I of G and attach each $v \in V \setminus I$ to an arbitrary neighbor in I. For each $v \in I$, let S_v denote the set consisting of v along with neighbors which have been attached to v. Let $\mathcal{S} = \{S_v \mid v \in I\}$ be the induced vertex partition of V. Since I is an independent set, in any realization Φ of G into $\mathbb{R}^d$, $\|\Phi(u) - \Phi(v)\|_2 > 1$ for all $u, v \in I, u \neq v$. From this observation, one can deduce the fact that $H := G[\mathcal{S}]$ has bounded growth rate. However, the sets S_v are not cliques, even though the subgraphs they induce, $H_v := G[S_v]$, have diameter at most 2. We know that each H_v has a partition into $O(1)$ cliques. For example, if G is a UDG, there exists a partition of H_v into at most 5 cliques. But, how to find a constant-sized partition of H_v? Noteworthy is the work of Raghavan and Spinrad [11] that computes a maximum cardinality clique of an input UBG, given without any geometric information; one can use their algorithm as a subroutine in the following greedy approach. Repeatedly find and remove a maximum size clique from H_v, until it becomes empty. Since we know that H_v can be partitioned into a constant c number of cliques, H_v contains a clique of size at least $|S_v|/c$ and therefore each step removes a $1/c$ fraction of

[2] Where it is clear from the context, we write $B(u, v)$ instead of $B_G(u, v)$.

vertices (or more) from H_v. This immediately implies that the greedy approach produces $O(\log n)$ cliques, where $n = |S_v|$. Unfortunately, this bound is tight and there is a simple example (see Figure 1(b)) showing that the greedy approach can lead to a clique-partition of S_v of size $\Omega(\log n)$. In Section 2 we present a polynomial time algorithm for partitioning H_v into $O(1)$ cliques.



(a) (b)

Fig. 1. (a) A realization of a UDG G, partitioned by a grid of $\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}}$ square cells. The cluster graph $G[\mathcal{C}]$ induced by the partition $\mathcal{C}$ is also shown. (b) A bad example for the greedy approach. Each A_i and each B_i is a set of 2^i points. A point in A_i is adjacent to all points in A_j, $1 \leq j \leq k$ and exactly the points in B_i. The adjacencies for points in B_i are symmetric. Thus the largest clique is $A_k \cup B_k$ and has size $2 \cdot 2^k$. Removing this clique leaves sets A_j and B_j, $1 \leq j \leq k-1$ intact. Thus the algorithm uses k cliques to cover about 2^{k+2} points, yielding a lower bound of $\Omega(\log n)$ on the size of the clique cover produced by the greedy algorithm.

Constructing Volume Respecting Embeddings. Let $H := G[\mathcal{C}]$ be the cluster graph of G constructed in the previous step. In the second step of our algorithm, we construct a *volume respecting embedding* of the shortest path metric of H. The notion of volume respecting embeddings was introduced by Feige [4] in the context of the minimum bandwidth problem. Let (X, d) be a metric space. An embedding $\Phi : X \mapsto \mathbb{R}^L$ is a *contraction* if $\|\Phi(u) - \Phi(v)\|_2 \leq d(u, v)$ for all $u, v \in X$. For a set T of k points in $\mathbb{R}^L$, define $\text{Evol}(T)$ to be the $(k-1)$-dimensional volume of the $(k-1)$-dimensional simplex spanned by T, computed using the ℓ_2 norm. Note that $\text{Evol}(T) = 0$ if T is affinely dependent. For any finite metric space (X, d), define $\text{Vol}(X)$ as $\sup_\Phi \text{Evol}(\Phi(X))$, where the supremum is over all contractions $\Phi : X \to \mathbb{R}^{|X|-1}$. Given an arbitrary metric space (X, d), a contraction $\Phi : X \to \mathbb{R}^L$ is called (k, D)-*volume respecting embedding* if for every size-k subset $S \subseteq X$, we have: $\text{Evol}(\Phi(S)) \geq \left(\frac{\text{Vol}(S)}{D^{k-1}}\right)$ Note that when $k = 2$, this condition reduces to $\|\Phi(u) - \Phi(v)\|_2 \geq d(u, v)/D$ for all $u, v \in X$. Thus, a volume respecting embedding is a generalization of the more commonly used notion of small distortion embeddings [9], in which only pairs of points are considered. A volume respecting embedding will be very useful for the last step our algorithm, in which a random projection of the vertices of H into $\mathbb{R}^2$, will be performed. In Section 3 we show how to construct a $(k, O(\sqrt{\log n}))$-volume respecting embedding of H. Here $k \geq 2$ and n is the number of vertices in H.

2 Constructing a Growth-Restricted Approximation

In this section, we show how to construct a clique partition $\mathcal{C} = \{C_1, C_2, \ldots\}$ of a given d-dimensional UBG G so that $G[\mathcal{C}]$ has constant growth rate. Consider the following algorithm.

`CLIQUE-PARTITION(`G`, `Φ`)`

1. Compute a maximal independent set (MIS) I of G.
2. Associate each vertex $u \in V \setminus I$ to a neighbor in I. For each $v \in I$, let S_v consist of v and its associated vertices.
3. Partition each vertex subset S_v into a constant number of cliques (this constant may depend on d).

Clearly, Steps (1)-(2) do not need any geometric information. In the following, we show how to implement the Step (3) of the `CLIQUE-PARTITION` algorithm, without using any geometric information about G. For this we make use of ideas due to Raghavan and Spinrad [11]. For simplicity, we discuss only the 2-dimensional case; extension to d-dimensional UBGs is straightforward.

Raghavan and Spinrad [11] present a "robust" algorithm for the problem of finding a maximum cardinality clique (henceforth, maximum clique) in a given UDG. Their algorithm is robust in the sense that it takes as input an arbitrary graph G and in polynomial time, (i) either returns a maximum clique in G or (ii) produces a certificate indicating that G is not a UDG. The key idea underlying the Raghavan-Spinrad algorithm is the existence of a superclass $\mathcal{G}$ of the class of UDGs such that in polynomial time one can determine if a given graph G is in $\mathcal{G}$ or not. Furthermore, for any G in $\mathcal{G}$ a maximum clique can be computed in polynomial time.

The superclass $\mathcal{G}$ is the set of all graphs that admit a *cobipartite neighborhood edge elimination ordering* (CNEEO). Given a graph $G = (V, E)$ and an edge ordering $L = (e_1, e_2, \ldots, e_m)$ of E, let $G_L[i]$ denote the spanning subgraph of G with edge set $\{e_i, e_{i+1}, \ldots, e_m\}$. For each edge $e_i = \{x, y\}$ define $N_L[i]$ to be the set of common neighbors of x and y in $G_L[i]$. An edge ordering $L = (e_1, e_2, \ldots, e_m)$ of $G = (V, E)$ is a CNEEO if for every edge e_i, $N_L[i]$ induces a cobipartite (i.e., the complement of a bipartite) graph. Raghavan and Spinrad prove three results: (1) If a graph G admits a CNEEO, then there is a simple greedy algorithm for finding a CNEEO of G. (2) Given a graph G and a CNEEO of G, a maximum clique in G can be found in polynomial time. (3) Every UDG admits a CNEEO. See Figure 2(a) for an illustration of why every UDG admits a CNEEO.

We now show how to use the CNEEO idea to implement Step (3) of the `CLIQUE-PARTITION` algorithm. Let $G_v := G[S_v]$ be the subgraph of G induced by S_v. Since G_v is also a UDG, it admits a CNEEO. We start with a simple lemma.

Lemma 1. *Let C be a clique in G_v and let $L = (e_1, e_2, \ldots, e_m)$ be a CNEEO of G_v. There is an i, $1 \leq i \leq m$, such that $N_L[i]$ contains C.*

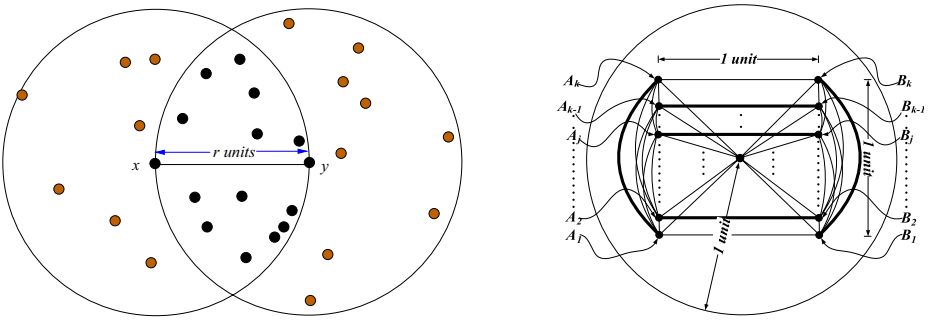


Fig. 2. (a) Suppose that edge $\{x, y\}$ has rank i in an ordering of the edges of G in non-increasing length order. The points in the common neighborhood of x and y in $G_L[i]$ are exactly those in the lune shown above. A point outside the lune may be a common neighbor of x and y in G, but not in $G_L[i]$. The diameter of the upper and lower halves of the lune are r and therefore the vertices in each half induce a clique in $G_L[i]$. Hence, $N_L[i]$ induces a cobipartite graph. (b) A sample run of `NBD-CLIQUE-PARTITION` on the "bad example" for the greedy algorithm. The 5 thick lines correspond (in some order) to the 5-edge sequence guessed by `NBD-CLIQUE-PARTITION` in Step (1). Among the guessed edges, the edge between A_1 and A_k, say e_A, and the edge between B_1 and B_k, say e_B, are critical because the common neighborhood of the endpoints of e_A is the clique $\bigcup_i A_i$ and similarly, the common neighborhood of the endpoints of e_B is the clique $\bigcup_i B_i$. Note that in the guess as highlighted by the thick lines, the algorithm will produce a clique partition with at most 10 cliques independent of the order in which the 5 edges are processed.

Proof. Let $e_i = \{x, y\}$ be the edge in $G_v[C]$ that occurs first in L. Recall the notation $N_L[i]$: this denotes the common neighborhood of the endpoints of edge e_i in the spanning subgraph of G_v containing only edges $e_i, e_{i+1}, \ldots$. Since e_i is the first edge in C, $N_L[i]$, contains C.

Recall that the closed neighborhood of a vertex in a UDG can be partitioned into at most 5 cliques. Let $C_1, C_2, \ldots, C_5$ be a clique partition of S_v. The implication of the above lemma is that even though we do not know the clique partition $C_1, C_2, \ldots, C_5$, we do know that for every CNEEO $L = (e_1, e_2, \ldots, e_m)$ of G_v, there is an edge e_i such that $N_L[i]$ can be partitioned into two cliques that cover C_1. This follows simply from the fact that L is a CNEEO and therefore the graph induced by $N_L[i]$ is cobipartite. This suggests an algorithm that starts by guessing an edge sequence $(f_1, f_2, \ldots f_5)$ of G_v. Then the algorithm computes L, a CNEEO of G_v. The algorithm's first guess is "good" if f_1 is the edge in C_1 that occurs first in L. Suppose this is the case and further suppose that f_1 has rank i in L. Then C_1 is contained in $N_L[i]$. Therefore, when $N_L[i]$ is deleted from G_v we have a graph, say G'_v, that can be partitioned into 4 cliques, namely $C_j \setminus N_L[i]$ for $j = 2, 3, 4, 5$. For each j, let C'_j denote $C_j \setminus N_L[i]$ and let L' be a CNEEO of G'_v. The algorithm's second guess, f_2, is "good" if f_2 is the edge in C'_2 that occurs first in L'. Letting i' denote the rank of f_2 in L', we see that

$N_{L'}[i']$ contains C'_2. Continuing in this manner we get a partition of G_v into 10 cliques. Below, the algorithm is described more formally.

```
NBD-CLIQUE-PARTITION(G_v)
1. for each 5-edge sequence (f_1, f_2, ... f_5) of E(G_v) do
2.         G_0 ← G_v
3.         for j ← 1 to 5 do
4.                 Compute a CNEEO L of G_{j-1}
5.                 i ← rank of f_j in L
6.                 Partition N_L[i] into two cliques C'_j and C''_j
7.                 G_j ← G_{j-1} \ N_L[i]
8.         if (G_5 = ∅) return {C'_j, C''_j | j = 1, 2, ..., 5}
```

The correctness of NBD-CLIQUE-PARTITION follows from the fact that there is some edge sequence $(f_1, f_2, \ldots f_5)$ of $E(G_v)$ for which G_5 is empty. Figure 2(b) shows a sample run of the algorithm. We state without proof the following lemma.

Lemma 2. *Algorithm* NBD-CLIQUE-PARTITION *partitions G_v into at most 10 cliques.*

If G_v has m edges then the number of guesses verified by the algorithm is $O(m^5)$. Since a CNEEO of a graph can be computed polynomial time (if it exists) and since a cobipartite can be partitioned into two cliques in linear time, we see that the algorithm NBD-CLIQUE-PARTITION runs in polynomial time. Thus Step (3) of the CLIQUE-PARTITION algorithm can be implemented by calling the NBD-CLIQUE-PARTITION algorithm for each vertex $v \in I$.

Let $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots\}$ be the clique partition of G produced by the algorithm CLIQUE-PARTITION. Let $H := G[\mathcal{C}]$. We now argue that H is growth-restricted. For each vertex $c \in V(H)$, there is a corresponding vertex v in the MIS I of G. Specifically, c corresponds to a clique in G that was obtained by partitioning S_v for some vertex $v \in I$. Recall that S_v consists of v along with some neighbors of v. Denote by $i(c)$ the vertex in I corresponding to $c \in V(H)$. Consider an arbitrary 2-dimensional realization Φ of G and for any pair of vertices $x, y \in V$, let $|xy|$ denote $\|\Phi(x) - \Phi(y)\|_2$. Let $B(v, r) = \{u \in V(H) \mid d_H(v, u) \leq r\}$ denote the closed ball of radius r, centered at v, with respect to shortest path distances in H. We present two structural lemmas pertaining to H.

Lemma 3. *For any $u, v \in V(H)$ and $r \geq 0$, if $u \in B(v, r)$ then $|i(u)i(v)| \leq 3r$.*

Proof. Consider two neighbors in H, x and y. Let C_x and C_y denote the cliques in G that were contracted into x and y respectively. Since x and y are neighbors in H, there are vertices $x' \in C_x$ and $y' \in C_y$ that are neighbors in G. Also, because of the way the cliques are constructed, x' is a neighbor of $i(x)$ and y' is a neighbor of $i(y)$. Since G is a UDG, by triangle inequality $|i(x)i(y)| \leq |i(x)x'| + |x'y'| + |y'i(y)| \leq 3$.

If $u \in B(v, r)$, then there is a uv-path P in H of length at most r. Corresponding to P there is a sequence of vertices in I starting with $i(u)$ and ending with

$i(v)$ such that consecutive vertices in this sequence are at most 3 units apart in any realization. Therefore, by triangle inequality $|i(u)i(v)| \leq 3r$.

Lemma 4. *There is a constant α such that for any $v \in V(H)$, $|B(v,r)| \leq \alpha \cdot r^2$.*

Proof. Let X be the number of vertices in $B(v,r)$. By the previous lemma, for each $u \in B(v,r)$, there is a vertex $i(u) \in I$ such that $i(u) \in Disk(i(v), 3r)$. Here $Disk(i(v), 3r)$ denotes the disk of radius $3r$ centered at vertex $i(v)$ in any realization of G. Also, by Lemma 2, there are at most 10 vertices in H that have the same corresponding vertex in I. Therefore, the number of vertices in $Disk(i(v), 3r)$ needs to be at least $X/10$. Any pair of vertices in I are more than one unit apart (in Euclidean distance) from each other. By the standard packing argument, this implies that the ball $Disk(i(v), 3r)$ can contain at most $4 \cdot (3r + 1/2)^2$ points in I. Therefore, $X/10 \leq 4(3r + 1/2)^2$ and hence for a constant α, we have $X \leq \alpha \cdot r^2$.

Lemma 4 can be generalized to higher dimensions to yield the following theorem.

Theorem 1. *There is a polynomial time algorithm that takes as input the combinatorial representation of a d-dimensional UBG $G = (V, E)$, and constructs a clique partition $\mathcal{C} = \{C_1, C_2, \ldots\}$ of G such that $G[\mathcal{C}]$ has growth rate $O(d)$.*

A side effect of our construction is that the constructed clique partition $\mathcal{C} = \{C_1, C_2, \ldots\}$ is a constant-factor approximation to the *minimum clique cover* for any UBG in fixed dimensional space. This follows from the fact that the size of any independent set is a lower bound on the size of a minimum clique cover and our solution produces a clique partition whose size is within a constant of the size of a maximal independent set.

Theorem 2. *There is a polynomial time algorithm that takes as input the combinatorial representation of a d-dimensional UBG $G = (V, E)$ for fixed d and constructs a clique partition $\mathcal{C} = \{C_1, C_2, \ldots\}$ of G, whose size is within a constant times the size of a minimum clique cover of G.*

3 Volume Respecting Embedding of Growth-Restricted Graphs

In this section, we show that by combining techniques due to Krauthgamer and Lee [6] with Rao's technique [12], we can derive a simple, combinatorial algorithm for constructing a $(k, O(\sqrt{\log n}))$-volume respecting embedding of any n-vertex growth-restricted graph and any $k \geq 2$. We emphasize the combinatorial nature of our algorithm because in earlier papers [13,10], this step involved solving an LP with exponentially many constraints via the ellipsoid method.

Rao [12] presents an algorithm for computing a $(k, O(\sqrt{\log n}))$-volume respecting embedding for graphs that exclude $K_{t,t}$ as a minor for a fixed t. A key step of Rao's embedding algorithm uses a graph decomposition due to Klein,

Plotkin, and Rao [5] that works for $K_{t,t}$-minor free graphs. In our case, the input graph is not guaranteed to be $K_{t,t}$-minor free for any t and therefore the Klein-Plotkin-Rao decomposition algorithm cannot be used. But, since our input graph is growth-restricted, we replace the Klein-Plotkin-Rao decomposition by the probabilistic decomposition due to Krauthgamer and Lee [6] that exploits the growth-restricted nature of the graph.

Let $G = (V, E)$ be a graph with growth rate ρ. Krauthgamer and Lee [6] present the following simple partitioning procedure for G, parameterized by an integer $r > 0$, that will be the first step of our algorithm. For any $M > 0$, let $\mathrm{T}exp(r, M)$ denote the probability distribution obtained by taking the continuous exponential distribution with mean r, truncating it at M, and rescaling the density function. The resulting distribution has density function $p(z) = \frac{e^{M/r}}{r(e^{M/r}-1)} \cdot e^{-z/r}$ for any $z \in (0, M)$. Let $V = \{v_1, v_2, \ldots, v_n\}$. For each vertex v_j, independently choose a radius r_j by sampling the distribution $\mathrm{T}exp(r, 8\rho r \ln r)$. Now define $S_j = B(v_j, r_j) \setminus \cup_{i=1}^{j-1} B(v_i, r_i)$. Thus S_j is the set of all vertices in the ball $B(v_j, r_j)$ that are not contained in any of the earlier balls $B(v_i, r_i)$, $1 \leq i \leq j-1$. It is clear that $\mathcal{C} = \{S_1, S_2, \ldots, S_n\}$ partitions G such that the weak diameter of each S_i is at most $16\rho r \ln r$.

Now we state a key lemma from Krauthgamer and Lee [6] that allows us to use Rao's technique. For any $u \in V$ and $x \geq 0$, let $\mathcal{E}_u^x$ denote the event that $B(u, x)$ is split between multiple clusters in $\mathcal{C}$.

Lemma 5. *Let $u \in V$, $r \geq 16\rho$, and $x \geq 0$. Then $\mathbf{Pr}[\mathcal{E}_u^x] \leq 10x/r$.*

The implication of this lemma is that for any vertex u, with probability at least a constant, a "large" ball (e.g., with radius $x = r/20$) centered at u is completely contained in one of the clusters in $\mathcal{C}$. This implication can be stated more precisely, as follows. For each cluster $C \in \mathcal{C}$, define the *boundary of C*, denoted ∂C, as the subset of vertices in C that have neighbors (in G) outside of C. Let B denote the set of all boundary vertices of all clusters in $\mathcal{C}$. Define a *δ-good vertex* to be a vertex that is at least δ hops away from any vertex in B. From Lemma 5 we can immediately derive the following.

Lemma 6. *Let $u \in V$ and $r \geq 16\rho$. With probability at least $\frac{1}{2}$, u is $\frac{r}{20}$-good.*

The above lemma (which is similar to Lemma 3 in Rao [12]) leads to the second stage of the algorithm. Consider the graph $G - B$. For each connected component Y in $G - B$, pick a *rate* α independently and uniformly at random from the interval $[1, 2]$. To each edge in Y, assign as weight the rate α corresponding to Y. Thus all edges in a connected component in $G - B$ have the same weight. To all other edges in G (i.e., those that are incident on vertices in B) assign the weight 0. Finally, to each vertex u in G assign a weight that is the length of a shortest path from u to a vertex in B. Note that the shortest paths are computed in the weighted version of G. Thus all vertices in B will get assigned a weight 0, whereas vertices in $G - B$ will get assigned a positive weight. In fact, it is easy to verify that the weight assigned to a vertex in $G - B$ is at least 1 and at most $16\rho r \ln r$. Exactly, as in Rao [12], we can derive the following lemma.

Lemma 7. *The weight of any δ-good node ranges uniformly over an interval I of length at least δ. Moreover, the choice is independent of anything in a different component.*

For each vertex v, the weight of v forms one coordinate of v. For each value of r, the above algorithm is repeated $\alpha k \log n$ times for a large enough constant α. Since Lemma 6 holds only for values of $r \geq 16\rho$, we repeat this the entire process for $r = 1, 16\rho, (16\rho)^2, \ldots, diam(G)$. For any constant ρ, this is essentially the same as using values of $r = 1, 2, 4, \ldots, diam(G)$ as Rao does [12]. Lemmas 6 and 7 are the two key results needed for the rest of Rao's analysis [12] to go through. Therefore, we get the following result.

Theorem 3. *For any $k \geq 2$, there is a polynomial time algorithm that constructs a $(k, O(\sqrt{\log n}))$-volume respecting embedding, with high probability, of any n vertex growth-restricted graph.*

4 Good Quality Embeddings of UBGs

In this section, we use the ideas developed in earlier sections to devise good quality embeddings of UBGs.

4.1 Constant Quality Embedding in Constant Dimensions

Levin, together with Linial, London, and Rabinovich [9], made a conjecture (Conjecture 8.2 in [9]) that is quite relevant to the best quality embedding problem. Let $\mathbb{Z}^d_\infty$ be the infinite graph with vertex set $\mathbb{Z}^d$ (i.e., the d-dimensional integral lattice) and an edge $\{u, v\}$ whenever $\|u - v\|_\infty = 1$. For any graph G, define $\dim(G)$ to be the smallest d such that G occurs as a (not necessarily induced) subgraph of $\mathbb{Z}^d_\infty$.

Conjecture 1. **[Levin, Linial, London, Rabinovich]** For any graph $G = (V, E)$ with growth rate ρ_G, G occurs as a (not necessarily induced) subgraph of $\mathbb{Z}^{O(\rho_G)}_\infty$. In other words, $\dim(G) = O(\rho_G)$.

Linial [8] introduced the following Euclidean analogue to this notion of dimensionality. For any graph G, define $\dim_2(G)$ to be the smallest d such that there is a mapping $\Phi : V \mapsto \mathbb{R}^d$ with the properties: (i) $\|\Phi(u) - \Phi(v)\|_2 \geq 1$ for all $u \neq v \in V$ and (ii) $\|\Phi(u) - \Phi(v)\|_2 \leq 2$ for all $\{u, v\} \in V$.

Lee and Krauthgamer [6] show that the specific bound on $\dim(G)$, mentioned in the above conjecture does not hold, by exhibiting a graph G for which $\dim(G) = \Omega(\rho_G \log \rho_G)$. They also prove a weaker form of the conjecture by showing that $\dim(G) = O(\rho_G \log \rho_G)$ for any graph G [6]. This proof relies on the Lovász Local Lemma, but can be turned into a polynomial time algorithm using standard techniques for turning proofs based on the Lovász Local Lemma into algorithms. Finally, they also prove that $\dim_2(G) = O(\rho_G \log \rho_G)$. This result, along with Theorem 1 leads to the following theorem.

Theorem 4. *There is a polynomial time algorithm, that takes as input a d-dimensional UBG and constructs an embedding of quality-2 in $O(d \log d)$ dimensional Euclidean space.*

4.2 $O(\log^{2.5} n)$ Quality Embedding in the Plane

In this section, we describe our complete algorithm. The first two steps of our algorithm are described in detail in the previous two sections. The last three steps respectively describe (i) a random projection into $\mathbb{R}^2$, (ii) a "rounding" step, and (iii) constructing an embedding of the original input graph G from the embedding of the cluster graph $G[\mathcal{C}]$. The random projection and the subsequent "rounding" step are essentially the same as in Vempala's algorithm [13].

Step 1. Construct a clique partition $\mathcal{C} = \{C_1, C_2, \ldots\}$ of the given UBG $G = (V, E)$ so that the induced cluster graph $H := G[\mathcal{C}]$ is growth-restricted. This is described in Section 2.

Step 2. Let $V(H) = \{v_1, v_2, \ldots, v_n\}$. Construct a $(\log n, O(\sqrt{\log n}))$-volume respecting embedding Φ of the shortest path metric of H, as described in Section 3. Let $u_i := \Phi(v_i)$ for $i = 1, 2, \ldots, n$.

Step 3. Choose 2 random lines, ℓ_1 and ℓ_2 (independently), passing through the origin. Project the point set $\{u_1, u_2, \ldots, u_n\}$ onto each of the two lines, mapping each u_i to $(u_i \cdot \ell_1, u_i \cdot \ell_2)$. Denote each $(u_i \cdot \ell_1, u_i \cdot \ell_2)$ by w_i.

Step 4. Discretize the plane into grid, with each cell having dimension $1/\sqrt{n} \times 1/\sqrt{n}$. Call each such grid cell an an *outer* grid cell. Let M be the maximum number of points w_i that fall in any cell after the random projection step. Subdivide each outer grid cell into a finer grid, with each cell having dimensions $1/\sqrt{n \cdot M} \times 1/\sqrt{n \cdot M}$. For each outer cell C, map all points that fall into C to grid points of the finer grid. Finally, scale up all grid points by a factor of $\sqrt{n \cdot M}$ along both dimensions so that each cell has unit width.

Step 5. Since every vertex v_i in H is associated with a clique C_i in G, all vertices in C_i are assigned the coordinates assigned to v_i in Step (4), to get the final embedding of G.

We skip the analysis of the above algorithm because it is similar to Vempala's analysis [13]. The final result we obtain is this.

Theorem 5 (Main Result). *With high probability, the quality of the embedding is $O(\log^{2.5} n)$.*

References

1. Aspnes, J., Goldenberg, D.K., Yang, Y.R.: On the computational complexity of sensor network localization. In: Nikoletseas, S.E., Rolim, J.D.P. (eds.) ALGOSENSORS 2004. LNCS, vol. 3121, pp. 32–44. Springer, Heidelberg (2004)
2. Basu, A., Gao, J., Mitchell, J.S.B., Sabhnani, G.: Distributed localization using noisy distance and angle information. In: MobiHoc '06: Proceedings of the seventh ACM international symposium on Mobile ad hoc networking and computing, pp. 262–273. ACM Press, New York, USA (2006)

3. Bruck, J., Gao, J., Jiang, A.(A).: Localization and routing in sensor networks by local angle information. In: MobiHoc '05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing, pp. 181–192. ACM Press, New York, NY, USA (2005)
4. Feige, U.: Approximating the bandwidth via volume respecting embeddings. J. Comput. Syst. Sci. 60(3), 510–539 (2000)
5. Klein, P., Plotkin, S.A., Rao, S.: Excluded minors, network decomposition, and multicommodity flow. In: STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pp. 682–690. ACM Press, New York, USA (1993)
6. Krauthgamer, R., Lee, J.R.: The intrinsic dimensionality of graphs. In: STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, pp. 438–447. ACM Press, New York, NY, USA (2003)
7. Kuhn, F., Moscibroda, T., O'Dell, R., Wattenhofer, M., Wattenhofer, R.: Virtual coordinates for ad hoc and sensor networks. Algorithmica (to appear 2006)
8. Linial, N.: Variation on a theme of Levin. In: Matoušek, J. (ed.) Open Problems, Workshop on Discrete Metric Spaces and their Algorithmic Applications (2002)
9. Linial, N., London, E., Rabinovich, Y.: The geometry of graphs and some of its algorithmic applications. Combinatorica 15, 215–245 (1995)
10. Moscibroda, T., O'Dell, R., Wattenhofer, M., Wattenhofer, R.: Virtual coordinates for ad hoc and sensor networks. In: DIALM-POMC '04: Proceedings of the 2004 joint workshop on Foundations of mobile computing, pp. 8–16. ACM Press, New York, NY, USA (2004)
11. Raghavan, V., Spinrad, J.: Robust algorithms for restricted domains. In: SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, pp. 460–467. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2001)
12. Rao, S.: Small distortion and volume preserving embeddings for planar and euclidean metrics. In: SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry, pp. 300–306. ACM Press, New York, USA (1999)
13. Vempala, S.: Random projection: A new approach to VLSI layout. In: FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science, pp. 389–395. IEEE Computer Society Press, Washington, DC, USA (1998)

Algorithms for Playing Games with Limited Randomness*

Shankar Kalyanaraman and Christopher Umans

Dept of Computer Science, California Institute of Technology, Pasadena, CA 91125
{shankar,umans}@cs.caltech.edu

Abstract. We study multiplayer games in which the participants have access to only limited randomness. This constrains both the algorithms used to compute equilibria (they should use little or no randomness) as well as the mixed strategies that the participants are capable of playing (these should be *sparse*). We frame algorithmic questions that naturally arise in this setting, and resolve several of them.

We give *deterministic* algorithms that can be used to find sparse ϵ-equilibria in zero-sum and non-zero-sum games, and a *randomness-efficient* method for playing repeated zero-sum games. These results apply ideas from derandomization (expander walks, and δ-independent sample spaces) to the algorithms of Lipton, Markakis, and Mehta [LMM03], and the online algorithm of Freund and Schapire [FS99].

Subsequently, we consider a large class of games in which sparse equilibria are known to exist (and are therefore amenable to randomness-limited players), namely games of small rank. We give the first "fixed-parameter" algorithms for obtaining approximate equilibria in these games. For rank-k games, we give a deterministic algorithm to find $(k+1)$-sparse ϵ-equilibria in time polynomial in the input size n and some function $f(k, 1/\epsilon)$. In a similar setting Kannan and Theobald [KT07] gave an algorithm whose run-time is $n^{O(k)}$. Our algorithm works for a slightly different notion of a game's "rank," but is fixed parameter tractable in the above sense, and extends easily to the multi-player case.

1 Introduction

Ever since Nash's work in 1951 [Nas51] and Morgenstern and von Neumann's treatise [MvN44] that introduced the subject, game theory has been one of the cornerstones of economic analysis. In recent years, game theory has increasingly become embedded into theoretical computer science. In particular, the field of algorithmic game theory has seen prolific contributions to areas such as auctions, mechanism design and congestion games.

Traditionally, designing algorithms for such games has been largely driven by the need to find equilibria *time-efficiently*. In this context, Chen and Deng [CD06] built on prior work [DP05, CD05, DGP06, GP06] and showed that the problem of finding a Nash equilibrium for the two-player game is complete for the PPAD class. This suggests that

* This research was supported by NSF grant CCF-0346991, BSF Grant 2004329, a Sloan Research Fellowship and an Okawa Foundation research grant.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 323–334, 2007.

a polynomial-time algorithm is unlikely to exist. With Teng, Chen and Deng [CDT06] showed further that finding $O(1/n^{\Theta(1)})$-approximate Nash equilibria is also PPAD-complete. In this paper, we study algorithms for finding equilibria and playing games *randomness-efficiently*. By "playing a game" we mean the actions a player must take to actually sample and play a strategy from a mixed strategy profile, which necessarily entails randomness. There is an immense body of literature in theoretical computer science devoted to addressing computational issues when limited or no randomness is available, and for the same reasons it makes sense to study games whose players have limited access to randomness. Moreover, limited randomness in games motivates some special classes of games that have arisen in other contexts thereby providing a viewpoint that we believe is helpful.

Limited randomness imposes two major constraints on players. Firstly, players with access to limited randomness are precluded from executing mixed strategies with large support since playing a mixed strategy with support size m requires at least $\log m$ bits of randomness (to sample a single strategy from the distribution). Secondly, they are limited to executing algorithms to help find optimal strategies or equilibria that use a small amount of randomness. We look at both such manifestations of limited randomness.

We first look at sparse strategies in single-round games. For the case of zero-sum games where both players have n available strategies, Lipton and Young [LY94] showed that a random sample of $O(\frac{\log n}{\epsilon^2})$ strategies was sufficient to approximate the value of the game by ϵ. Lipton, Markakis and Mehta [LMM03] extended this to ϵ-equilibria for two-player nonzero-sum games. Indeed, they gave a randomized procedure that produced small-support strategies for an ϵ-equilibrium when given a Nash equilibrium with possibly large support. In the following theorem, we derandomize this procedure by using random walks on expander graphs:

Theorem 1. *Let $G = (R, C, n)$ be a two-player game, and let (p^*, q^*) be a Nash equilibrium for G. For every $\epsilon > 0$, there is a* deterministic *procedure P running in time poly$(|G|)^{1/\epsilon^2}$ such that the pair $(P(G, p^*, 1), P(G, q^*, 2))$ is an $O(\frac{\log n}{\epsilon^2})$-sparse 4ϵ-equilibrium for G.*

This can be viewed as a *deterministic* "sparsification" procedure for ϵ-equilibria in general two player games. In zero-sum games one can find optimal strategies efficiently, and as a result, we obtain a deterministic polynomial time algorithm to find sparse ϵ-equilibria for zero-sum games:

Corollary 1. *Let $G = (R, C, n)$ be a two-player* zero-sum *game. For every $\epsilon > 0$, there is a* deterministic *procedure running in time poly$(|G|)^{1/\epsilon^2}$ that outputs a pair (p^*, q^*) that an $O(\frac{\log n}{\epsilon^2})$-sparse ϵ-equilibrium for G.*

We point out that Freund and Schapire [FS96] obtained a similar result using an adaptive multiplicative-weight algorithm (discussed below) with $poly(|G|, 1/\epsilon)$ running time. We obtain the corollary above using a very different proof technique that flows from well-known derandomization techniques.

In single-round games a randomness-limited player requires sparse strategies, but in multiple-round games, we would like to be able to "reuse" randomness across rounds. This is an orthogonal concern to that of reducing randomness within a single round.

Freund and Schapire [FS99] proposed an adaptive online algorithm for a T-round two-player zero-sum game with n strategies available to each. Executing the mixed strategies produced by their algorithm uses $\Omega(T \log n)$ bits of randomness over T rounds, in the worst case. By making use of almost-pairwise independence, we show how to reuse randomness across rounds: it is possible to make do with just $O(\log n + \log\log T + \log(1/\epsilon))$ bits and achieve close to the same quality of approximation as in [FS99].

Theorem 2. *Let M be the $n \times n$-payoff matrix for a two-player zero-sum T-round game with entries in $\{0,1\}$. For any $\epsilon < 1/2$ and constant δ, there exists an online randomized algorithm R using $O(\log n + \log\log T + \log(1/\epsilon))$ random bits with the following property: for any arbitrary sequence $Q_1, \ldots, Q_T$ of mixed strategies played (adaptively) by the column player over T rounds, R produces a sequence of strategies $S_1, \ldots, S_T$ such that with probability at least $1 - \delta$:*

$$\frac{1}{T}\sum_{i=1}^{T} M(S_i, Q_i) \leq \frac{1}{T}\min_{P}\sum_{i=1}^{T} M(P, Q_i) + O\left(\sqrt{\frac{\log n}{T}} + \epsilon\right)$$

As we have discussed, players with limited access to randomness can only achieve equilibria that are sparse. We saw before that in the general setting, we are able to "sparsify" deterministically if we are willing to settle for ϵ-equilibria. The sparsity cannot in general be less than $\log n$, though, so we are motivated to consider broad classes of games in which even sparser equilibria are guaranteed to exist. Perhaps the simplest example is a 2-player games in which one player has only k available strategies, while the other player has $n \gg k$ available strategies. The results in Lipton et al. [LMM03] imply there is a Nash equilibrium in this game with support size $k+1$. This is somewhat unsatisfying – it means that in a two-player game one player may need to choose from less-sparse strategies than his opponent (i.e. requiring slightly *more* randomness) to achieve equilibrium. Theorem 3 rectifies this asymmetry by showing that k-sparse strategies suffice for the opposing player.

Theorem 3. *Let $G = (R, C, k, n)$ be a two-player game. Given p^* for which there exists a q^* such that (p^*, q^*) is a Nash equilibrium, we can compute in deterministic polynomial time q' for which (p^*, q') is a Nash equilibrium and $|supp(q')| \leq k$.*

We also give a deterministic polynomial time algorithm to compute such a limited-support strategy for one player, given the k-sparse strategy of the other. We can extend this further to the multiplayer case to show that for an ℓ-player game where players 1 through $\ell - 1$ have $k_1, \ldots, k_{\ell-1}$ pure strategies, respectively, the ℓ-th player need only play a $(\prod_{i=1}^{\ell-1} k_i)$-sparse strategy to achieve equilibrium (these bounds are tight).

Perhaps the most significant technical contribution in this paper pertains to a generalization of the "unbalanced" games that we saw above, namely, games of small rank. This is a broad class of games for which sparse equilibria are known to exist. For 2-player games with rank-k payoff matrices, Lipton et al. describe an enumeration algorithm that finds $(k+1)$-sparse strategies in $O(n^{k+1})$ time. We improve dramatically on this bound but we must relax the problem in two ways: first, we compute an ϵ-equilibrium rather than an exact one; second we require that the payoff matrices be presented as a low-rank decomposition, with entries up to precision B (limited precision makes sense since we are only considering ϵ-equilibria).

Theorem 4. *Let $G = (R, C, n)$ be a two player game such that R and C have rank at most k. Furthermore, let $R = R_1R_2$, $C = C_1C_2$ be a decomposition of R, C with R_1, R_2, C_1, C_2 containing integer entries in $[-B, B]$. Then, for every $\epsilon > 0$, there is a deterministic procedure P running in time $(4B^2k/\epsilon)^{2k}poly(|G|)$ that returns a 4ϵ-Nash equilibrium (p, q) with $|supp(p)|, |supp(q)| \leq k + 1$.*

To the best of our knowledge, Theorem 4 provides the first efficient "fixed-parameter" algorithm to this problem in the sense that the running time is polynomial in the input size n and some function $f(k, 1/\epsilon, B)$. The closest parallel to our result is by Kannan and Theobald [KT07] who consider a somewhat different definition of "rank" for two-player games: in their definition, the *sum* of the payoff matrices is required to have small rank. In that case, they present an algorithm that finds an ϵ-equilibrium in a rank k 2-player game in $O(n^{2k+o(1)}B^2)$ time. Their algorithm relies on results of Vavasis for solving indefinite quadratic programs [Vav92] and does not seem to generalize to $\ell > 2$ players. Our algorithm is (arguably) simpler, and moreover, it easily generalizes to $\ell > 2$ players, where small rank games still are guaranteed to have sparse equilibria. In the ℓ-player setting, we give an $O(((2B)^\ell k\ell/\epsilon)^{k\ell(\ell-1)})\text{poly}(n^\ell)$ time deterministic procedure that computes such a sparse ϵ-equilibrium, when the payoff tensors are presented as a rank-k decomposition with entries up to precision B.

All of the algorithms for low-rank games rely on enumerating potential equilibria distributions in a basis dictated by the small rank decomposition. This seems like a technique that may be useful for algorithmic questions regarding low-rank games beyond those we have considered in this paper.

2 Preliminaries

Definition 1. *For a finite strategy set S we define $\mathbf{\Delta}(S)$ to be the set of probability distributions over S, i.e, all vectors $p = (p_s)_{s\in S}$ satisfying $\sum_{s\in S} p_s = 1$ with each $p_s \in [0, 1]$. A* **mixed strategy** *p is an element of $\Delta(S)$. The* **support** *of a mixed strategy $p \in \Delta(S)$ is the set $supp(p)$ given by $supp(p) = \{s \in S | p_s > 0\}$. A mixed strategy p is k-***sparse** *if $|supp(p)| = k$.*

In this paper, we will concern ourselves with games that can be specified by payoff matrices (or tensors) whose entries denote the payoff upon playing the corresponding strategy tuple. We will also assume, unless otherwise specified, that these entries are bounded and can be scaled to lie in $[-1, 1]$.

Definition 2. *An ℓ-***player finite game** *G is a tuple $(T_1, \ldots, T_\ell, n_1, n_2, \ldots, n_\ell)$ where T_i is the $(n_1 \times \ldots \times n_\ell)$ ℓ-dimensional payoff tensor with $T_i(s_1, \ldots, s_\ell)$ denoting the payoff to player i when the pure strategy ℓ-tuple $(s_1, \ldots, s_\ell)$ is played in the game.*

For ease of presentation, in the rest of this paper we will often restrict ourselves to ℓ-player games where $n_1 = n_2 = \cdots = n_\ell = n$, which we denote by $G = (T_1, \ldots, T_\ell, n)$. We often refer to players by their payoff tensors. For example, for the two-player game $G = (R, C, n)$ we will refer to the row player as R and the column player as C. All vectors are thought of as row vectors.

Definition 3. *In an ℓ-player game $G = (T_1, T_2, \ldots, T_\ell, n_1, n_2, \ldots, n_\ell)$, we denote by $T_i(p_1, \ldots, p_\ell)$ the payoff to the i-th player when the ℓ players play mixed strategies $p_1, \ldots, p_\ell$, i.e.,*

$$T_i(p_1, \ldots, p_\ell) = \sum_{i_1 \in [n_1], \ldots, i_\ell \in [n_\ell]} p_{i_1} p_{i_2} \ldots p_{i_\ell} T_i(i_1, i_2, \ldots, i_\ell).$$

If we substitute some $a \in [n_j]$ for p_j we understand that to mean the distribution that places weight 1 on a and 0 everywhere else.

Definition 4. *Let $G = (T_1, \ldots, T_\ell, n_1, \ldots, n_\ell)$ be an ℓ-player. An l-tuple $(p_1^*, \ldots, p_\ell^*)$ with each $p_i^* \in \Delta([n_i])$ is an ϵ-**equilibrium** for G if: for every i and every $p \in \Delta([n_i])$,*

$$T_i(p_1^*, \ldots, p_{i-1}^*, p, p_{i+1}^*, \ldots, p_\ell^*) \leq T_i(p_1^*, \ldots, p_{i-1}^*, p_i^*, p_{i+1}^*, \ldots, p_\ell^*) + \epsilon.$$

A **Nash equilibrium** *is an ϵ-equilibrium for $\epsilon = 0$.*

Let G be an ℓ-player game. It is well-known that given the *supports* of the ℓ different p_i^* in a Nash equilibrium, one can find the actual *distributions* by linear programming. We will use a similar fact repeatedly (we defer the proof to a longer version of the paper [KU07]):

Lemma 1. *Let $G = (T_1, T_2, \ldots, T_\ell, n)$ be an ℓ-player game, and let $(p_1^*, p_2^*, \ldots, p_\ell^*)$ be a Nash equilibrium. Given G and $p_1^*, p_2^*, \ldots, p_{\ell-1}^*$ one can find a distribution q in deterministic polynomial time for which $(p_1^*, p_2^*, \ldots, p_{\ell-1}^*, q)$ is also a Nash equilibrium.*

3 Sparsifying Nash Equilibria Deterministically

In this section we give deterministic algorithms for "sparsifying" Nash equilibria (in the process turning them into ϵ-equilibria). In this way, a player with limited access to randomness, but who has access to an equilibrium mixed strategy, is able to produce a small strategy that can then be played.[1] We prove our results in detail for the two player case, and then sketch the (easy) extension to multiple players in the longer version of the paper [KU07]. Lipton et al. showed the following result:

Theorem 5 (Lipton et al. [LMM03]). *Let $G = (R, C, n)$ be a two-player game, and let (p^*, q^*) be a Nash equilibrium for G. There is a polynomial-time randomized procedure P such that with probability at least 1/2, the pair $(P(G, p^*), P(G, q^*))$ is an $O(\log n/\epsilon^2)$-sparse ϵ-equilibrium for G.*

The algorithm P is very simple: it amounts to sampling uniformly from the given equilibrium strategy. The analysis applies Chernoff bounds to show that the sampled strategies present the opposing players with approximately (within ϵ) the same weighted row- and column- sums, and hence constitute an ϵ-equilibrium. In our setting, since the

[1] The question of *how* the player may obtain an equilibrium mixed strategy is a separate and well-studied topic, but not the focus of this work.

players have limited randomness they cannot afford the above sampling (it requires at least as much randomness as simply playing (p^*, q^*)), so we derandomize the algorithm using an expander walk. Before proving Theorem 1, we will give a convenient characterization of ϵ-equilibrium:

Lemma 2. *Let $G = (R, C, n)$ be a 2-player game. Define*

$$T_p = \{i | (pC)_i \geq \max_r (pC)_r - \epsilon\}$$
$$S_q = \{j | (Rq^T)_j \geq \max_t (Rq^T)_t - \epsilon\}.$$

If supp$(p) \subseteq S_q$ and supp$(q) \subseteq T_p$, then (p, q) is an ϵ-equilibrium for G.

We will use the Chernoff bound for random walks on an expander:

Theorem 6 (Gillman [Gil93]). *Let H be an expander graph with second largest eigenvalue λ and vertex set V, and let $f : V \to [-1, 1]$ be arbitrary with $E[f] = \mu$. Let $X_1, X_2, \ldots, X_t$ be the random variables induced by first picking X_1 uniformly in V and $X_2, \ldots, X_t$ by taking a random walk in H from X_1. Then*

$$\Pr\left[\left|\frac{1}{t}\sum_i f(X_i) - \mu\right| > \delta\right] < e^{-O((1-\lambda)\delta^2 t)}.$$

Proof (of Theorem 1). When we are given G and p^*, we perform the following steps. First, construct a multiset S of $[n]$ for which uniformly sampling from S approximates p^* to within ϵ/n. This can be done with $|S| \leq O(n/\epsilon)$. Denote by $\tilde{p}$ the distribution induced by sampling uniformly from S. We identify S with the vertices of a constant-degree expander H, and we can sample $S' \subseteq S$ by taking a walk of length $t = O(\log n/\epsilon^2)$ steps in H. Note that this requires $O(\log |S| + O(t)) = O(\log n/\epsilon^2)$ random bits. Let p' be the probability distribution induced by sampling uniformly from S'. By Theorem 6 (and using the fact that C has entries in $[-1, 1]$), for each fixed i,

$$\Pr[|(p'C)_i - (\tilde{p}C)_i| \geq \epsilon] \leq e^{-O(\epsilon^2 t)} < 1/n. \quad (1)$$

By a union bound $|(p'C)_i - (\tilde{p}C)_i| \leq \epsilon$ for all i with non-zero probability. This condition can be checked given G, p^*, and so we can derandomize the procedure completely by trying all choices of the random bits used in the expander walk.

When we are given G and q^*, we perform essentially the same procedure (with respect to R and q^*), and in the end we output a pair p', q' for which $|(p'C)_i - (\tilde{p}C)_i| \leq \epsilon$ for all i and $|(Rq'^T)_j - (R\tilde{q}^T)_j| \leq \epsilon$ for all j. We claim that any such (p', q') is a 4ϵ-equilibrium, assuming (p^*, q^*) is an equilibrium. Using the fact that C, R have entries in $[-1, 1]$, and the fact that our multiset approximations to p^*, q^* have error at most ϵ/n in each coordinate, we obtain that $|(\tilde{p}C)_i - (p^*C)_i| \leq \epsilon$ for all i and $|(R\tilde{q}^T)_j - (Rq^{*T})_j| \leq \epsilon$ for all j. Define (as in Lemma 2)

$$T_{p'} = \{i | (p'C)_i \geq \max_i (p'C)_i - 4\epsilon\}$$
$$S_{q'} = \{j | (Rq'^T)_j \geq \max_j (Rq'^T)_j - 4\epsilon\}.$$

Now, $w \in \text{supp}(p')$ implies $w \in \text{supp}(p^*)$ which implies $(Rq^{*T})_w = \max_j (Rq^{*T})_j$ (since (p^*, q^*) is a Nash equilibrium). From above we have that $\max_j (Rq'^T)_j \leq \max_j (Rq^{*T})_j + 2\epsilon$ and that $(Rq'^T)_w \geq (Rq^{*T})_w - 2\epsilon$. So $(Rq'^T)_w \geq \max_j (Rq'^T)_j - 4\epsilon$, and hence w is in $S_{q'}$. We conclude that $\text{supp}(p') \subseteq S_{q'}$. A symmetric argument shows that $\text{supp}(q') \subseteq T_{p'}$. Applying Lemma 2, we conclude that (p', q') is a 4ϵ-equilibrium as required and this completes the proof. □

Since an equilibrium can be found efficiently by Linear Programming in the two player zero-sum case, we obtain Corollary 1. We extend the algorithm above to make it work for games involving three or more players.

Theorem 7. *Let $G = (T_1, T_2, \ldots, T_\ell, n)$ be an ℓ-player game, and let $(p_1^*, p_2^*, \ldots, p_\ell^*)$ be a Nash equilibrium for G. For every $\epsilon > 0$, there is a* deterministic *procedure P running in time poly$(|G|)^{1/\epsilon^2}$, such that the tuple $(P(G, p_1^*, 1), P(G, p_2^*, 2), \ldots, P(G, p_\ell^*, \ell))$ is an $O((\ell \log n)/\epsilon^2)$-sparse 4ϵ-equilibrium for G.*

4 Limited Randomness in Repeated Games

So far we have looked at optimizing the amount of randomness needed in single-round games where players execute their strategies only once. In this section, we investigate multiple-round games and in particular, the adaptive multiplicative weight algorithm of Freund and Schapire [FS99] for which we describe randomness-efficient modifications. In particular, we show that by using almost-pairwise independent random variables it is possible to achieve close to the same quality of approximation as in their work.

Note that we make crucial use of the full power of [FS99] – i.e., their guarantee still holds if the column player changes his play in response to the particular randomness-efficient sampling being employed by the row player.

Proof (of Theorem 2). Our randomized online algorithm R is a modification of Freund and Schapire's multiplicative-weight adaptive algorithm [FS99]. For a game with payoff matrix M where both players have n strategies belonging to a strategy-set S, and for a sequence of mixed strategies $(P_1, P_2, \ldots, P_T)$ over T rounds for the first player described by

$$P_{i+1}(s) = \left(\frac{\beta^{M(s,Q_t)}}{\sum_s p_i(s) \beta^{M(s,Q_t)}} \right) p_i(s) \tag{2}$$

where $Q_1, \ldots, Q_T$ are arbitrary (adaptive) mixed strategies of the column player and $\beta = 1/(1+\sqrt{2 \log n / T})$, the Freund-Schapire algorithm offers the following guarantee on the expected payoff over T rounds:

Lemma 3 (Freund and Schapire [FS99])

$$\frac{1}{T} \sum_{t=1}^{T} M(P_t, Q_t) \leq \min_P \frac{1}{T} \sum_{t=1}^{T} M(P, Q_t) + O\left(\sqrt{\frac{\log n}{T}} \right) \tag{3}$$

Running the Freund-Schapire algorithm requires $\Omega(T \log n)$ random bits in order to select a strategy from each distribution but we can optimize on this by using almost pairwise independent random variables.

Definition 5. *[AGHP92] Let $Z_n \subseteq \{0,1\}^n$ be a sample space and $X = x_1 \dots x_n$ be chosen uniformly from Z_n. Z_n is (ρ, k)-***independent** *if for any positions $i_1 < i_2 < \dots < i_k$ and any k-bit string $t_1 \dots t_k$, we have*

$$\left| \Pr_X [x_{i_1} x_{i_2} \dots x_{i_k} = t_1 \dots t_k] - 2^{-k} \right| \leq \rho$$

Alon et al. [AGHP92] give efficient constructions of (ρ, k)-independent random variables over $\{0,1\}^n$ that we can suitably adapt to obtain T $(\rho, 2)$-independent random variables $S_1, \dots, S_T$ over a larger alphabet of size $O(n/\epsilon)$ using $O(\log n + \log(1/\rho) + \log(1/\epsilon) + \log \log T)$ random bits.

Note that as we did in the proof of Theorem 1 in Section 3, we can approximate any distribution P_t by a uniform distribution S_t drawn from a multiset of size $O(n/\epsilon)$ that approximates P_t to within ϵ/n and suffer at most $O(\epsilon)$ error. Therefore, under the uniform distribution over vertices $s \in S_i$ for all $i = 1, \dots, T$: $|M(P_i, Q_i) - E[M(S, Q_i)]| \leq O(\epsilon)$. The following lemma is the key step in our derandomization.

Lemma 4. *Let $S_1, \dots, S_T$ be $(\rho, 2)$-independent random variables. Then, for any δ:*

$$\Pr_{S_1,\dots,S_T} \left[\frac{1}{T} \sum_{i=1}^{T} M(S_i, Q_i) \geq \frac{1}{T} E \left[\sum_{i=1}^{T} M(S_i, Q_i) \right] + \sqrt{\frac{1}{\delta} \left(\frac{1}{T} + \frac{2\rho n^2}{\epsilon^2} \right)} \right] \leq \delta$$

Setting $\rho = O(\delta \epsilon^6 / n^2)$ in Lemma 4 gives us that with probability at least $1 - \delta$ over the choice of randomness of $S_1, \dots, S_T$

$$\frac{1}{T} \sum_{i=1}^{T} M(S_i, Q_i) \leq \frac{1}{T} E \left[\sum_{i=1}^{T} M(S_i, Q_i) \right] + \epsilon \leq \frac{1}{T} \sum_{i=1}^{T} M(P_i, Q_i) + O(\epsilon)$$

Finally by application of Lemma 3 we have with probability at least $1 - \delta$

$$\frac{1}{T} \sum_{i=1}^{T} M(S_i, Q_i) \leq \frac{1}{T} \min_P \sum_{i=1}^{T} M(P, Q_i) + O\left(\sqrt{\frac{\log n}{T}} \right) + O(\epsilon) \tag{4}$$

Note that by our choice of ρ, we require $O(\log n + \log \log T + \log(1/\epsilon))$ random bits. This completes the proof of the theorem. □

5 Unbalanced Games

In this section we look at what happens when one of the players (perhaps as a consequence of having limited randomness) is known to have very few – say k – available strategies, while the other player has $n \gg k$ available strategies. In such a game does there exist a k-sparse strategy for the second player? We prove that this is indeed the case. The main technical tool we use is a constructive version of Carathéodory's Theorem which we state below.

Theorem 8 (Carathéodory's Theorem, constructive version). *Let $v_1, v_2, \ldots, v_n$ be vectors in a k-dimensional subspace of $\mathbb{R}^m$ where $n \geq k+1$, and suppose*

$$v = \sum_{i=1}^{n} \alpha_i v_i \quad \text{with} \sum_i \alpha_i = 1 \text{ and } \alpha_i \geq 0 \text{ for all } i \tag{5}$$

Then there exist $\alpha'_1, \ldots, \alpha'_n$ for which $v = \sum_{i=1}^{n} \alpha'_i v_i$ with $\sum_i \alpha'_i = 1$ and $\alpha'_i \geq 0$ for all i, and $|\{i : \alpha'_i > 0\}| \leq k+1$. Moreover the α'_i can be found in polynomial time, given the α_i and the v_i.

Due to paucity of space, we defer the proof of Theorem 3 and its generalization to ℓ players to the longer version [KU07]. The main idea is to argue that for a Nash equilibrium distribution (p^*, q^*), $u = Rq^{*T}$ lies in a $(k-1)$-dimensional subspace. We then apply Theorem 8 to express u as a convex combination $R'q'^T$ where $|supp(q')| \leq k$. This gives us the required Nash equilibrium (p^*, q').

6 Finding Sparse ϵ-Equilibria in Low-Rank Games

We now consider games of rank k, which is a significant generalization of the "unbalanced" games in the previous section. Indeed, rank k games are perhaps the most general class of games for which sparse equilibria are guaranteed to exist. In this section we give algorithms to compute sparse ϵ-equilibria in this setting. Lipton et al. showed that there *exist* $(k+1)$-sparse Nash equilibria in this setting and this implies an enumeration algorithm to find an equilibrium in time approximately $n^{k+1}\text{poly}(|G|)$. Our algorithm shows that the problem is "fixed parameter tractable" [Ces05, DF99, DFS97] where ϵ, the rank k and precision B are the parameters.

We first look at the two-player case. Since we are computing ϵ-equilibria, we only expect the game specification to be given up to some fixed precision. We will be working with rank k matrices M expressed as $M_1 M_2$ (where M_1 is a $n \times k$ matrix and M_2 is a $k \times n$ matrix). Such a decomposition can be found efficiently via basic linear algebra. In the following theorem we take M_1 and M_2, specified up to fixed precision, as our starting point.[2] As the example in §6.1 illustrates, such a decomposition is already available for many natural games. Our convention for expressing fixed precision entries will be to require them to be integers in the range $[-B, B]$ for a bound B.

Proof (of Theorem 4). Note that the payoff to the row-player when (p, q) is the strategy tuple for the game which is given by pRq^T can now be written as $pR_1R_2q^T$ and likewise for the column player. The first step in our algorithm is to "guess" a collection of vectors to within $\delta = \epsilon/(2Bk)$ precision. We describe the "correct" guess relative to an (arbitrary) Nash equilibrium (p^*, q^*) for G. Let $p^{*'} = p^*C_1, q^{*'} = R_2q^{*T}$. Note that from our choice of C_1, R_2 it holds that $p^{*'}, q^{*'}$ satisfy $-B \leq p^{*'}_i, q^{*'}_i \leq B; i = 1, \ldots, k$. The algorithm is as follows:

[2] We note that computing M_1, M_2 of fixed precision such that M_1M_2 approximates M is not necessarily always possible or straightforward. We state our theorem in this way to avoid these complications, a detailed discussion of which would be beyond the scope of this paper.

1. Guess a $\tilde{p}'$ such that for all $i = 1, \ldots, k$ $|p_i^{*'} - \tilde{p}'_i| \leq \delta$. Similarly, guess $\tilde{q}'$ such that for all $i = 1, \ldots, k$ $|q_i^{*'} - \tilde{q}'_i| \leq \delta$.
2. Let $\alpha_s = (\tilde{p}'C_2)_s$ and $\beta_t = (R_1\tilde{q}'^T)_t$. Set $S = \{s | \max_r \alpha_r - 2\epsilon \leq \alpha_s \leq \max_r \alpha_r\}$ and $T = \{t | \max_r \beta_r - 2\epsilon \leq \beta_t \leq \max_r \beta_r\}$.
3. Find a feasible solution $\bar{p}$ to the following linear program

$$|(\bar{p}C_1)_j - \tilde{p}'_j| \leq \delta; \ j = 1, \ldots, k \tag{6}$$

$$\bar{p}_i \geq 0; \ i = 1, \ldots, n \tag{7}$$

$$\bar{p}_i = 0; \ i \notin T \tag{8}$$

$$\sum_{i=1}^{n} \bar{p}_i = 1 \tag{9}$$

   and a feasible solution $\bar{q}$ to the analogous linear program in which the first set of constraints is

$$|(R_2\bar{q}^T)_j - \tilde{q}'_j| \leq \delta; j = 1, \ldots, k.$$

4. $v = \bar{p}C_1$ is a convex combination of the rows of C_1, all of which lie in a k-dimensional subspace. From Carathéodory's Theorem (Theorem 8), we can find $\hat{p}$ with $\text{supp}(\hat{p}) \subseteq \text{supp}(p)$ for which $|\text{supp}(\hat{p})| \leq k + 1$ and $v = \hat{p}C_1$.
5. Similarly $u = R_2\bar{q}^T$ is a convex combination of the columns of R_2, all of which lie in a k-dimensional subspace. Applying Theorem 8 again, we find $\hat{q}$ with $\text{supp}(\hat{q}) \subseteq \text{supp}(q)$ for which $|\text{supp}(\hat{q})| \leq k + 1$ and $u = R_2\hat{q}^T$.
6. Return $\hat{p}, \hat{q}$.

Correctness follows from the next two claims, whose proofs are in the longer version [KU07]:

Claim. A feasible solution to the linear programs formulated in step 3 of the algorithm exists.

Claim. $(\hat{p}, \hat{q})$ as returned by the algorithm is a 4ϵ-equilibrium.

It is not hard to see that the run-time of the algorithm is $(4B^2k/\epsilon)^{2k}$ taking into account the enumerations of $\tilde{p}'$. The applications of Theorem 8 also take poly$(|G|)$ and so the running time is as claimed. □

For the case of three or more players, obtaining approximate Nash equilibria for low-rank games does not seem to have been studied previously. Furthermore, the previously best known algorithms for low-rank games [KT07] do not seem to extend to more than two players. Theorem 9 provides an extension of the result above to the multi-player case.

Theorem 9. *Let $G = (T_1, \ldots, T_\ell, n)$ be an ℓ-player game, and suppose we are given a k-decomposition of $T_i = (C_{i1}, \ldots, C_{i\ell})$ where each of the C_{ij} is an $n \times k$ matrix with integer values in $[-B, B]$ for $i, j = 1, \ldots, \ell$. Then for every $\epsilon > 0$, there is a deterministic procedure P running in time*

$$\left((2B)^\ell k\ell/\epsilon\right)^{k(\ell-1)\ell} \textit{poly}(|G|)$$

that returns a 4ϵ-Nash equilibrium $(p_1, p_2, \ldots, p_\ell)$ with $|\textit{supp}(p_i)| \leq 1 + \ell k$ for all i.

6.1 An Example of Games with Known Low-Rank Tensor Decomposition

Many natural games are specified implicitly (rather than by explicitly giving the tensors) by describing the payoff function, which itself is often quite simple. In such cases, the tensor ranks may be significantly smaller than n, and moreover, a low-rank decomposition into components with bounded entries can often be derived from the payoff functions.

One prominent example is simple ℓ-player congestion games as discussed in game theory literature [FPT04, Pap05]. Such a game is based on a graph $\mathcal{G}(V, E)$ with n vertices and m edges. Each player's strategy set corresponds to a subset $S_p \subseteq 2^E$, the set of all subsets of edges. We define the payoff accruing to some strategy l-tuple $(s_1, \ldots, s_\ell)$ as $U(s_1, \ldots, s_\ell) = -\sum_e c_e(s_1, \ldots, s_\ell)$ where $c_e(s_1, \ldots, s_\ell) = |\{i | e \in s_i, 1 \leq i \leq \ell\}|$ is thought of as the *congestion* on paths $s_1, \ldots, s_\ell$. Let $G = (T_1, \ldots, T_\ell, N = 2^m)$ be the game corresponding to the situation described above where for $i = 1, \ldots, \ell$ and strategy tuple $(s_1, \ldots, s_\ell)$, $T_i(s_1, \ldots, s_\ell) = -\sum_e c_e(s_1, \ldots, s_\ell)$. We defer proof of the following theorem to the longer version of the paper [KU07].

Theorem 10. *For $i = 1, \ldots, \ell$ T_i as defined above is of rank at most ℓm. Furthermore, an explicit ℓm-decomposition $(C_{i1}, C_{i2}, \ldots, C_{i\ell})$ for T_i exists where C_{ij} are $n \times k$ matrices with entries in $\{-1, 0, 1\}$.*

7 Conclusion

There are many other interesting questions that are raised by viewing game theory through the lens of requiring players to be randomness-efficient. In this paper, we have framed some of the initial questions that arise and have provided answers to several of them. In particular, we have exploited the extensive body of work in derandomization to construct deterministic algorithms for finding sparse ϵ-equilibria (which can be played with limited randomness), and for playing repeated games while reusing randomness across rounds. The efficient fixed-parameter algorithms we describe for finding ϵ-equilibria in games of small rank significantly improve over the standard enumeration algorithm, and to the best of our knowledge, they are the first such results for games of small rank.

The notion of *resource-limited* players has been an extremely useful one in game theory, and we think that it is an interesting and natural question in this context to consider the case in which the limited computational resource is *randomness*. These considerations expose a rich and largely untapped area straddling complexity theory and game theory.

Acknowledgments. We are grateful to the anonymous referees for their valuable comments and suggestions.

References

[AGHP92] Alon, N., Goldreich, O., Hastad, J., Peralta, R.: Simple constructions of almost k-wise independent random variables. Random Structures and Algorithms 3, 289–304 (1992)

[CD05] Chen, X., Deng, X.: 3-NASH is PPAD-complete. Electronic Colloquium on Computational Complexity (ECCC) (134) (2005)

[CD06] Chen, X., Deng, X.: Settling the Complexity of Two-Player Nash Equilibrium. In: Foundations of Computer Science (FOCS 2006), pp. 261–272. IEEE Computer Society Press, Los Alamitos (2006)

[CDT06] Chen, X., Deng, X., Teng, S.-H.: Computing Nash Equilibria: Approximation and Smoothed Complexity. In: Foundations of Computer Science (FOCS 2006), pp. 603–612. IEEE Computer Society Press, Los Alamitos (2006)

[Ces05] Cesati, M.: Compendium of parameterized problems (2005), http://bravo.ce.uniroma2.it/home/cesati/research/compendium/

[DF99] Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)

[DFS97] Downey, R.G., Fellows, M.R., Stege, U.: Parameterized complexity: A framework for systematically confronting computational intractability. In: Proceedings of the First DIMATIA Symposium (1997)

[DGP06] Daskalakis, C., Goldberg, P.W., Papadimitriou, C.H.: The complexity of computing a Nash equilibrium. In: Symposium on Theory of Computing (STOC 2006), pp. 71–78 (2006)

[DP05] Daskalakis, C., Papadimitriou, C.H.: Three-player games are hard. Electronic Colloquium on Computational Complexity (ECCC) (139) (2005)

[FPT04] Fabrikant, A., Papadimitriou, C.H., Talwar, K.: The complexity of pure-strategy equilibria. In: Symposium on Theory of Computing (STOC 2004), pp. 604–612 (2004)

[FS96] Freund, Y., Schapire, R.: Game theory, on-line prediction and boosting. In: COLT 1996, pp. 325–332 (1996)

[FS99] Freund, Y., Schapire, R.: Adaptive game playing using multiplicative weights. Games and Economic Behavior 29, 79–103 (1999)

[Gil93] Gillman, D.: A Chernoff bound for random walks on expander graphs. In: Foundations of Computer Science (FOCS 1993), pp. 680–691. IEEE, Los Alamitos (1993)

[GP06] Goldberg, P.W., Papadimitriou, C.H.: Reducibility among equilibrium problems. In: Symposium on Theory of Computing (STOC 2006), pp. 61–70 (2006)

[KT07] Kannan, R., Theobald, T.: Games of fixed rank: A hierarchy of bimatrix games. In: ACM-SIAM Symposium on Discrete Algorithms, ACM Press, New York (2007)

[KU07] Kalyanaraman, S., Umans, C.: Algorithms for playing games with limited randomness (2007), http://www.cs.caltech.edu/~shankar/pubs/ku07-apglr.pdf

[LMM03] Lipton, R.J., Markakis, E., Mehta, A.: Playing large games using simple strategies. In: EC '03: Proceedings of the 4th ACM conference on Electronic commerce, pp. 36–41. ACM Press, New York, NY, USA (2003)

[LY94] Lipton, R.J., Young, N.E.: Simple strategies for large zero-sum games with applications to complexity theory. In: STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, pp. 734–740. ACM Press, New York, USA (1994)

[MvN44] Morgenstern, O., von Neumann, J.: Theory of Games and Economic Behavior. Princeton University Press, Princeton (1944)

[Nas51] Nash, J.F.: Non-cooperative games. Annals of Mathematics 54, 286–295 (1951)

[Pap05] Papadimitriou, C.H.: Computing Correlated Equilibria in Multi-Player Games. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing, pp. 49–56 (May 2005)

[Vav92] Vavasis, S.A.: Approximation algorithms for indefinite quadratic programming. Math. Program. 57, 279–311 (1992)

Approximation of Partial Capacitated Vertex Cover*

Reuven Bar-Yehuda[1], Guy Flysher[2,**], Julián Mestre[3,***], and Dror Rawitz[4,†]

[1] Department of Computer Science, Technion, Haifa 32000, Israel
reuven@cs.technion.ac.il
[2] Google Inc., Building 30, MATAM, Haifa 31905, Israel
guyfl@google.com
[3] Department of Computer Science, University of Maryland, College Park, MD 20742, USA
jmestre@cs.umd.edu
[4] School of Electrical Engineering, Tel Aviv University, Tel Aviv 69978, Israel
rawitz@eng.tau.ac.il

Abstract. We study the *partial capacitated vertex cover problem* (PCVC) in which the input consists of a graph G and a covering requirement L. Each edge e in G is associated with a *demand* $\ell(e)$, and each vertex v is associated with a *capacity* $c(v)$ and a *weight* $w(v)$. A feasible solution is an assignment of edges to vertices such that the total demand of assigned edges is at least L. The weight of a solution is $\sum_v \alpha(v)w(v)$, where $\alpha(v)$ is the number of copies of v required to cover the demand of the edges that are assigned to v. The goal is to find a solution of minimum weight. We consider three variants of PCVC. In PCVC with *separable demands* the only requirement is that total demand of edges assigned to v is at most $\alpha(v)c(v)$. In PCVC with *inseparable demands* there is an additional requirement that if an edge is assigned to v then it must be assigned to one of its copies. The third variant is the unit demands version. We present 3-approximation algorithms for both PCVC with separable demands and PCVC with inseparable demands and a 2-approximation algorithm for PCVC with unit demands. We show that similar results can be obtained for PCVC in hypergraphs and for the *prize collecting* version of capacitated vertex cover. Our algorithms are based on a unified approach for designing and analyzing approximation algorithms for capacitated covering problems. This approach yields simple algorithms whose analyses rely on the local ratio technique and sophisticated charging schemes.

1 Introduction

The problems. Given a graph $G = (V, E)$, a *vertex cover* is a subset $U \subseteq V$ such that each edge in G has at least one endpoint in U. In the *vertex cover problem*, we are given a graph G and a weight function w on the vertices, and our

* Research supported in part by REMON — Israel 4G Mobile Consortium.
** Research was done while the author was a graduate student at the Technion.
*** Supported by the University of Maryland Dean's Dissertation Fellowship.
† Research was done while the author was a postdoc at CRI, University of Haifa.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 335–346, 2007.

goal is to find a minimum weight vertex cover. Vertex cover is NP-hard [1], and cannot be approximated within a factor of $10\sqrt{5} - 21 \approx 1.36$, unless P=NP [2]. On the positive side, there are several 2-approximation algorithms for vertex cover (see [3,4] and references therein).

The *capacitated vertex cover problem* (CVC) is an extension of vertex cover in which each vertex u has a *capacity* $c(u) \in \mathbb{N}$ that determines the number of edges it may cover, i.e., u may cover up to $c(u)$ incident edges. Multiple copies of u may be used to cover additional edges, provided that the weight of u is counted for each copy (*soft capacities*). A feasible solution is an assignment of every edge to one of its endpoints.

A *capacitated vertex cover* is formally defined as follows. An *assignment* is a function $A : V \to 2^E$. That is, for every vertex u, $A(u) \subseteq E(u)$, where $E(u)$ denotes the set of edges incident on u. An edge e is said to be *covered by* A (or simply *covered*) if there exists a vertex u such that $e \in A(u)$. Henceforth, we assume, w.l.o.g., that an edge is covered by no more than one vertex. An assignment A is a *cover* if every edge is covered by A, i.e., if $\bigcup_{u \in V} A(u) = E$. The *multiplicity* (or number of copies) of a vertex u with respect to an assignment A is the smallest integer $\alpha(u)$ for which $|A(u)| \leq \alpha(u)c(u)$. The weight of a cover A is $w(A) = \sum_u \alpha(u)w(u)$. Note that the presence of zero-capacity vertices may render the problem infeasible, but detecting this is easy: the problem is infeasible if and only if there is an edge whose two endpoints have zero capacity. Also note that vertex cover is the special case where $c(u) = \deg(u)$ for every vertex u.

In a more general version of CVC we are given a *demand* or *load* function $\ell : E \to \mathbb{N}$. In the *separable* edge demands case $\alpha(u)$ is the number of copies of u required to cover the total demand of the edges in $A(u)$, i.e., $\alpha(u)$ is the smallest integer for which $\sum_{e \in A(u)} \ell(e) \leq \alpha(u)c(u)$. In the case of *inseparable* demands there is an additional requirement that if an edge is assigned to u then it must be assigned to one of its copies. Hence, if the demand of an edge is larger than the capacity of both its endpoints, it cannot be covered. Clearly, given an assignment A, this additional requirement may only increase $\alpha(u)$. (Each vertex faces its own *bin packing* problem.) Hence, if all edges are coverable in the inseparable demands sense, then the optimum value for CVC with separable demands is not larger than the optimum value for CVC with inseparable demands.

Another extension of vertex cover is the *partial vertex cover problem* (PVC). In PVC the input consists of a graph G and an integer L and the objective is to find a minimum weight subset $U \subseteq V$ that covers at least L edges. PVC extends vertex cover, since in vertex cover $L = |E|$. In a more general version of PVC we are given edge *demands*, and the goal is to find a minimum weight subset U that covers edges whose combined demand is at least L.

In this paper we study the *partial capacitated vertex cover problem* (PCVC) that extends both PVC and CVC. In this problem we are asked to find a minimum weight capacitated vertex cover that covers edges whose total demand is at least L. We consider three variants of PCVC: PCVC with *separable demands*, PCVC with *inseparable demands*, and PCVC with unit demands.

Motivation. CVC was proposed by Guha et al. [5], who were motivated by a problem from the area of bioinformatics. They described a chip-based technology, called GMID (Glycomolecule ID), that is used to discover the connectivity of building blocks of glycoproteins. Given a set of building blocks (vertices) and their possible connections (edges), the goal is to identify which connections exist between the building blocks in order to discover the variant of the glycoprotein in question. Given a block B and a set S that consists of k of B's neighbors, GMID can reveal which of the building blocks from S are connected to B. The problem of minimizing the number of GMID operations needed to cover the required information graph is exactly CVC with uniform capacities. Since not all possible combinations of ties between building blocks may appear, it is sometimes sufficient to probe only a fraction of the edges in the information graph. In this case the problem of minimizing the number of GMID operations is PCVC with uniform capacities. (For details on the structure of glycoproteins see [6]).

CVC can be seen as a *capacitated facility location problem*, where the edges are the clients and the vertices are the capacitated facilities. For instance, CVC in hyper-graphs can be used to model a cellular network planning problem, in which we are given a set of potential base stations that are supposed to serve clients located in receiving areas. Each such area corresponds to a specific subset of the potential base stations. We are supposed to assign areas (edges) to potential base stations (vertices) and to locate transmitters with limited bandwidth capacity in each potential base station that will serve the demand of receiving areas that are assigned to it. Our goal is to assign areas to base stations so as to minimize transmitter costs. In terms of the above cellular network planning problem, PCVC is the case where we are required to cover only a given percentage of the demands.

Related work. Capacitated covering problems date back to Wolsey [7] who presented a greedy algorithm for weighted set cover with hard capacities that achieves a logarithmic approximation ratio. Guha et al. [5] presented a primal-dual 2-approximation algorithm for CVC. (A local ratio version was given in [4].) They also gave a 3-approximation algorithm for CVC with separable edge demands. Both algorithms can be extended to hyper-graphs in which the maximum size of an edge is Δ. The resulting approximation ratios are Δ and $\Delta + 1$. Guha et al. [5] also studied CVC in trees. They obtained polynomial time algorithms for the unit demands case and for the unweighted case, and proved that CVC with separable demands in trees is weakly NP-hard. Gandhi et al. [8] presented a 2-approximation algorithm for CVC using an LP-rounding method called *dependent rounding*. Chuzhoy and Naor [9] presented a 3-approximation algorithm for unweighted CVC with hard capacities, and proved that the weighted version of this problem is as hard to approximate as set cover. Gandhi et al. [10] improved the approximation ratio for unweighted CVC with hard capacities to 2.

Partial set cover was studied by Kearns [11], who proved that the performance ratio of the greedy algorithm is at most $2H_m + 3$, where H_m is the mth harmonic number. Slavík [12] showed that it is actually bounded by H_m. Bshouty and Burroughs [13] obtained the first 2-approximation algorithm for PVC. Bar-Yehuda [14] presented a local ratio 2-approximation algorithm for PVC

with demands. His algorithm extends to a Δ-approximation algorithm in hyper-graphs in which the maximum size of an edge is Δ. Gandhi et al. [15] presented a different algorithm with the same approximation ratio for PVC with unit demands in hyper-graphs that is based on a primal-dual analysis. Building on [15], Srinivasan [16] obtained a polynomial time $(2 - \Theta(\frac{1}{d}))$-approximation algorithm for PVC, where d is the maximum degree of a vertex in G. Halperin and Srinivasan [17] developed a $(2 - \Theta(\frac{\ln \ln d}{\ln d}))$-approximation algorithm for PVC.

Recently, Mestre [18] presented a primal-dual 2-approximation algorithm for PCVC with unit demands that is based on the 2-approximation algorithm for CVC from [5] and on the 2-approximation algorithm for PVC from [15].

Our results. We present constant factor approximation algorithms to several variants of PCVC. Our algorithms are based on a unified approach for designing and analyzing approximation algorithms for capacitated covering problems. This approach yields simple algorithms whose analyses rely on the local ratio technique and sophisticated charging schemes. Our method is inspired by the local ratio interpretations of the approximation algorithms for CVC from [5] and the local ratio 2-approximation algorithm for PVC from [14].

We present a 3-approximation algorithm for PCVC with separable demands whose analysis is one of the most sophisticated local ratio analyses in the literature. In the full version of the paper we present a 3-approximation algorithm for PCVC with inseparable demands. As far as we know, this algorithm is the first 3-approximation algorithm for CVC with inseparable demands. We also present a 2-approximation algorithm for PCVC with unit demands which is much simpler and more intuitive than Mestre's algorithm [18]. Note that our algorithm is not a local ratio manifestation of Mestre's algorithm. While his algorithm relies on the algorithm for PVC from [15], our algorithm extends the algorithm for PVC from [14]. In the full version of the paper we also show that our algorithms can be extended to PCVC in hyper-graphs, where the maximum size of an edge is Δ. The approximation ratios for separable demands, inseparable demands, and unit demands are $\Delta + 1$, $\Delta + 1$, and Δ, respectively. Finally, we show that the same ratios can be obtained for *prize collecting* vertex cover in hyper-graphs.

2 Preliminaries

Notation and terminology. Given an undirected graph $G = (V, E)$, let $E(u)$ be the set of edges incident on u, and let $N(u)$ be the set of u's neighbors. Given an edge set $F \subseteq E$ and a demand (or load) function ℓ on the edges of G, we denote the total demand of the edges in F by $\ell(F)$, i.e., $\ell(F) = \sum_{e \in F} \ell(e)$. We define $\deg(u) = \ell(E(u))$. Hence, in the unit demands case $\deg(u)$ is the degree of u, i.e., $\deg(u) = |E(u)| = |N(u)|$.

We define $\tilde{c}(u) = \min\{c(u), \deg(u)\}$. This definition may seem odd, since we may assume that $c(u) \leq \deg(u)$ for every vertex u in the input graph. However, our algorithms repeatedly remove vertices from the given graph, and therefore we may encounter a graph in which there exists a vertex where $\deg(u) < c(u)$. We define $b(u) = \min\{c(u), \deg(u), L\} = \min\{\tilde{c}(u), L\}$. $b(u)$ can be seen as the

covering potential of u in the current graph. A single copy of u can cover $c(u)$ of the demand if $\deg(u) \geq c(u)$, but if $\deg(u) < c(u)$, we cannot cover more than a total of $\deg(u)$ of the demand. Moreover, if L is smaller than $\tilde{c}(u)$, we have nothing to gain from covering more than L.

The *support* of an assignment A is the set of vertices $V(A)=\{u : A(u) \neq \emptyset\}$. We denote by $E(A)$ the set of edges covered by A, i.e., $E(A)=\cup_u A(u)$. We define $|A|=|E(A)|$ and $\ell(A)=\ell(E(A))$. Note that in the unit demand case $\ell(A)=|A|$.

Small, medium, and large edges. Given a PCVC instance, we refer to an edge $e=(u,v)$ as *large* if $\ell(e) > c(u), c(v)$. If $\ell(e) \leq c(u), c(v)$ it is called *small*. Otherwise, e is called *medium*.

Consider the case of PCVC with inseparable demands. Since a large edge cannot be assigned to a single copy of one of its endpoints, we may ignore large edges in this case. Notice that the instance may contain a medium edge $e=(u,v)$ such that $c(u) < \ell(e) \leq c(v)$. If there exist such an edge e, then we may add a new vertex u_e to the graph, where $w(u_e)=\infty$ and $c(u_e)=\ell(e)$, and connect e to u_e instead of to u. We refer to this operation as an *edge detachment*. Since $e \notin A(u_e)$ in any solution A of finite weight, it follows that we may assume, w.l.o.g, that all edges are small when the demands are inseparable.

A similar problem may happen in the separable demands case when there exists a vertex u such that $c(u)=0$. We assume that there are no such vertices. If there exists such a vertex u we simply define $c(u)=1$ and $w(u)=\infty$.

Local ratio. The *local ratio technique* [19] is based on the Local Ratio Theorem, which applies to problems of the following type: given a non-negative weight vector w and a set of feasibility constraints $\mathcal{F}$, find a vector $x \in \mathbb{R}^n$ that minimizes the inner product $w \cdot x$ subject to the set of constraints $\mathcal{F}$.

Theorem 1 (Local Ratio [20]). *Let $\mathcal{F}$ be a set of constraints and let w, w_1, and w_2 be weight vectors such that $w = w_1 + w_2$. Then, if x is r-approximate with respect to $(\mathcal{F}, w_1)$ and with respect to $(\mathcal{F}, w_2)$, for some r, then x is also an r-approximate solution with respect to $(\mathcal{F}, w)$.*

3 Partial Capacitated Covering with Separable Demands

We present a 3-approximation algorithm for PCVC with separable demands. At the heart of our scheme is Algorithm **PCVC**, which is inspired by the local ratio interpretation of the approximation algorithms for CVC from [5] and the local ratio approximation algorithm for PVC from [14].

In the description of the algorithm, we use a function called **Uncovered** that, given a vertex u, returns an uncovered edge e incident on u with maximum demand, i.e., it returns an edge $e \in E(u) \setminus A(u)$ such that $\ell(e) \geq \ell(e')$ for every $e' \in E(u) \setminus A(u)$. (If $A(u)=E(u)$ it returns NIL).

First, observe that there are $O(|V|)$ recursive calls. Hence, the running time of the algorithm is polynomial.

Consider the recursive call made in Line 5. In order to distinguish between the assignment obtained by this recursive call and the assignment returned in

Algorithm 1. **PCVC**(V, E, w, L)

1: **if** $L = 0$ **then return** $A(v) \leftarrow \emptyset$ for all $v \in V$
2: **if** there exists $x \in V$ and $e_x \in E(x)$ s.t. $\ell(e_x) > \max\{L, c(x)\}$ **then**
   return $A(x) \leftarrow \{e_x\}$ and $A(v) \leftarrow \emptyset$ for all $v \neq x$
3: **if** there exists $u \in V$ s.t. $\deg(u) = 0$ **then return** **PCVC**$(V \setminus \{u\}, E, w, L)$
4: **if** there exists $u \in V$ s.t. $w(u) = 0$ **then**
5: $A \leftarrow$ **PCVC**$(V \setminus \{u\}, E \setminus E(u), w, \max\{L - \deg(u), 0\})$
6: **if** $V(A) = \emptyset$
7: **while** $\ell(A(u)) < L$ **do**
$A(u) \leftarrow A(u) \cup \{$**Uncovered**$(u)\}$
8: **while** $(A(u) \neq E(u))$ and $(\ell(A(u)) + \ell($**Uncovered**$(u)) < c(u))$ **do**
$A(u) \leftarrow A(u) \cup \{$**Uncovered**$(u)\}$
9: **else**
10: **if** $\ell(A) < L$ **then** $A(u) \leftarrow E(u)$
11: **return** A
12: Let $\varepsilon = \min_{u \in V}\{w(u)/b(u)\}$
13: Define the weight functions $w_1(v) = \varepsilon \cdot b(v)$, for every $v \in V$, and $w_2 = w - w_1$
14: **return** **PCVC**(V, E, w_2, L)

Line 11, we denote the former by A' and the latter by A. Assignment A' is for $G' = (V', E')$, and we denote the corresponding parameters α', $\deg'$, $\tilde{c}'$, b', and L'. Similarly, we use α, deg, $\tilde{c}$, b, and L, for the parameters of A and $G = (V, E)$.

Lemma 1. *Algorithm* ***PCVC*** *computes a partial capacitated vertex cover.*

Proof. First, notice that we assign only uncovered edges. We prove that $\ell(A) \geq L$ by induction on the recursion. Consider the base case. If the recursion ends in Line 1 then $\ell(A) = 0 = L$. Otherwise the recursion ends in Line 2 and $\ell(A) = \ell(e_x) > L$. In both cases the solution is feasible. For the inductive step, if the recursive call was made in Line 3 or in Line 14, then $\ell(A) \geq L$ by the inductive hypothesis. If the recursive call was made in Line 5, then $\ell(A') \geq \max\{L - \deg(u), 0\}$ by the inductive hypothesis. If $V(A') = \emptyset$, then $\deg(u) \geq L$, and therefore $\ell(A(u)) \geq L$. Otherwise, if $V(A') \neq \emptyset$ then there are two options. If $\ell(A') \geq L$, then $A = A'$ and we are done. If $\ell(A') < L$, the edges in $E(u)$ are assigned to u, and since their combined demand is $\deg(u)$ it follows that $\ell(A) = \ell(A') + \deg(u) \geq L' + \deg(u) = L$. □

The assignment returned by Algorithm **PCVC** is not 3-approximate in general. If there exists a very large edge whose covering is enough to attain feasibility then Line 2 will choose to cover the edge no matter how expensive its endpoints may be. Nevertheless, we can still offer the following slightly weaker guarantee.

Theorem 2. *If the recursion of Algorithm* ***PCVC*** *ends in Line 1 then the assignment returned is 3-approximate. Otherwise, if the recursion ends in Line 2, assigning edge e_x to vertex x, then the solution returned is 3-approximate compared to the cheapest solution that assigns e_x to x.*

The proof of the theorem is given in Sect. 4.

Observe that Line 2 is never executed, if all edges are small. Thus, by Theorem 2, the algorithm computes 3-approximations when the instance contains only small edges. It can be shown that without Line 2 the algorithm may fail to provide a 3-approximation if the instance contains medium or large edges.

Based on Theorem 2 it is straightforward to design a 3-approximation algorithm using Algorithm **PCVC**. First **PCVC** is run on the input instance. Suppose the recursion ends in Line 2 with edge e_x assigned to vertex x. The assignment found is considered to be a *candidate solution* and set aside. Then the instance is modified by detaching edge e_x from x. These steps are repeated until an execution of Algorithm **PCVC** ends its recursion in Line 1 with the empty assignment. At the end, a candidate solution with minimum cost is returned.

Theorem 3. *There exists a 3-approximation algorithm for* PCVC *with separable demands.*

Proof. First, notice that once an edge is detached from an endpoint it cannot be detached from the same endpoint again. Thus, Algorithm **PCVC** is executed $O(|E|)$ times, and the overall running time is polynomial.

We argue by induction on the number of calls to **PCVC** that at least one of the candidate solutions produced is 3-approximate. For the base case, the first call to **PCVC** ends with the empty assignment and by Theorem 2 the assignment is 3-approximate. For the inductive step, the first run ends with edge e_x assigned to vertex x. If there exists an optimal solution that assigns e_x to x then by Theorem 2 the assignment is 3-approximate. Otherwise the problem of finding a cover in the new instance, where e_x is detached from x, is equivalent to the problem on the input instance. By inductive hypothesis, one of the later calls is guaranteed to produce a 3-approximate solution. We ultimately return a candidate solution with minimum cost, thus the theorem follows. □

4 Analysis of Algorithm PCVC

In the next two sections we pave the way for the proof of Theorem 2. Our approach involves constructing a subtle charging scheme by which at each point we have at our disposal a number of *coins* that we distribute between the vertices. The charging scheme is later used in conjunction with the Local Ratio Theorem to relate the cost of the solution produced by the algorithm to the cost of an optimal solution. We describe two changing schemes depending on how the recursion ends. We use the following observations in our analysis.

Observation 1. *Suppose a recursive call is made in Line 5. Then,* $\alpha(v) = \alpha'(v)$ *for every* $v \in V \setminus \{u\}$.

Observation 2. $\alpha(v)\tilde{c}(v) \leq 2\deg(v)$. *Moreover, if* $\alpha(v) > 1$ *then,* $\alpha(v)\tilde{c}(v) \leq 2\ell(A(v))$

Proof. Since $\tilde{c}(v) \leq \deg(v)$, the first claim is trivial if $\alpha(v) \leq 2$. If $\alpha(v) > 1$ then $\deg(v) > c(v)$, and therefore $\alpha(v)\tilde{c}(v) = \alpha(v)c(v) < \ell(A(v)) + c(v) \leq 2\ell(A(v)) \leq 2\deg(v)$. □

4.1 The Recursion Ends with the Empty Assignment

In this section we study what happens when the recursion of Algorithm **PCVC** ends in Line 1. Let $\$(v)$ be the number of coins that are given to v. A charging scheme $\$ is called *valid* if $\sum_v \$(v) \leq 3L$. We show that if the assignment A was computed by Algorithm **PCVC** then there is a way to distribute $3L$ coins so that $\$(v) \geq \alpha(v)b(v)$. The proof of the next lemma contains a recursive definition of our charging scheme. We denote by $v_0, v_1, \ldots$ the vertices of $V(A)$ in the order they join the set.

Lemma 2. *Let A be an assignment that was computed by Algorithm **PCVC** for a graph $G = (V, E)$ and a covering demand L. Furthermore, assume the recursion ended in Line 1. Then, one of the following conditions must hold:*

1. *$V(A) = \emptyset$. Also, the charging scheme $\$(v) = 0$ for every $v \in V$ is valid.*
2. *$V(A) = \{v_0\}$, $\alpha(v_0) = 1$, and $\ell(A(v_0)) \geq \frac{1}{2}c(v_0)$. Also, the charging scheme $\$(v_0) = 3L$ and $\$(v) = 0$ for every $v \neq v_0$ is valid.*
3. *$V(A) = \{v_0\}$, $\alpha(v_0) = 2$, and $\ell(A(v_0)) - \ell(e_0) < L$, where e_0 is the last edge that was assigned to v_0. Also, the charging scheme $\$(v_0) = 3L$ and $\$(v) = 0$ for every $v \neq v_0$ is valid.*
4. *$V(A) = \{v_0, v_1\}$, $\alpha(v_0) = 1$, $\alpha(v_1) \geq 2$, and $\ell(A(v_0)) \geq \frac{1}{2}c(v_0)$. Also, there exists a valid charging scheme $\$ such that $\$(v_0) \geq L$, $\$(v_0) \geq 2\ell(A(v_0)) - (\ell(A) - L)$, $\$(v_1) \geq \alpha(v_1)c(v_1)$, and $\$(v) = 0$ for every $v \neq v_0, v_1$.*
5. *$\alpha(v_0) = 1$, $\ell(A(v_0)) < \frac{1}{2}c(v_0)$, $\alpha(v) \geq 2$ for every $v \in V(A) \setminus \{v_0\}$, and $L > \deg(v_0) - \ell(A(v_0))$. Also, there exists a valid charging scheme $\$ such that $\$(v_0) \geq 2L + \ell(A(v_0)) - \ell(A)$, $\$(v) \geq \alpha(v)c(v)$ for every $v \in A(v) \setminus \{v_0\}$, and $\$(v) = 0$ for every $v \notin V(A)$.*
6. *$V(A) \neq \emptyset$. Also, there exists a valid charging scheme $\$ such that $\$(v) \geq \alpha(v)\tilde{c}(v)$ for every $v \in V$.*

Proof sketch. We prove the lemma by induction on the length of the creation series of the solution. Specifically, we assume that one of the conditions holds and prove that the augmented solution always satisfies one of the conditions. The possible transitions from condition to condition are given in Fig. 1. First, at the base of the induction the computed solution is the empty assignment and therefore Cond. 1 holds. For the inductive step, there are three possible types of recursive calls corresponding to Line 3, Line 5, and Line 14 of Algorithm **PCVC**. The calls in Line 3 and Line 14 do not change the assignment that is returned by the recursive call, nor does it change b for any vertex. Thus, if it satisfies one of the conditions it continues to satisfy them. The only correction is needed in the case of Line 3, where we need to extend the corresponding charging scheme by assigning $\$(u) = 0$. For the rest of the proof we concentrate on recursive calls that are made in Line 5. We consider a solution A' that was computes by the recursive call, and denote by $\$'$ the charging scheme that corresponds to A'.

Consider a recursive call in which A' satisfies Cond. 1. In this case, $V(A) = \{u\}$. There are four possible options:

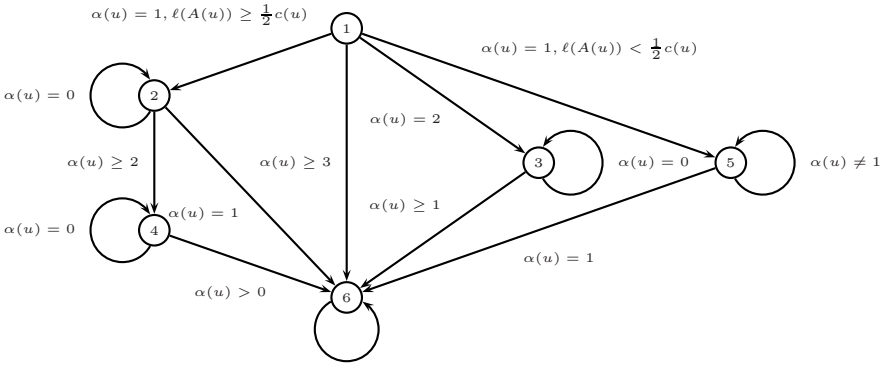


Fig. 1. Possible transitions between the conditions

$(1 \to 5)$ If $\alpha(u) = 1$ and $\ell(A(u)) < c(u)/2$, then we claim that $\ell(A(u)) = \deg(u)$. Observe that edges are added to $A(u)$ in a non-increasing order of demands in Lines 7 and 8. Hence, $\ell(A(u)) < c(u)/2$ implies that $A(u) = E(u)$. It follows that $\deg(u) - \ell(A(u)) = 0 < L$. Consider the charging scheme $\$(u) = 3L$ and $\$(v) = 0$ for every $v \neq u$. $\$(u) = 3L \geq 2L + \ell(A(v_0)) - \ell(A)$ since $\ell(A) = \ell(A(v_0))$. Hence, Cond. 5 holds.

$(1 \to 2)$ If $\alpha(u) = 1$ and $\ell(A(u)) \geq c(u)/2$, then Cond. 2 holds.

$(1 \to 3)$ If $\alpha(u) = 2$, then by the construction of $A(u)$ (Line 7), we know that $\ell(A(u)) - \ell(e_u) < L$, where e_u is the last edge that was added to $A(u)$. Hence, Cond. 3 holds.

$(1 \to 6)$ If $\alpha(u) \geq 3$, let e_u be the last edge assigned to u. Observe that $\ell(e_u) \leq L$, since Line 2 was not executed, and $\ell(A(u)) < L + \ell(e_u)$ due to Line 7. Hence, $\ell(A(u)) < 2L$. Furthermore, note that $\tilde{c}(u) = c(u)$, since $\deg(u) > c(u)$. Let the charging scheme be $\$(u) = 3L$ and $\$(v) = 0$ for every $v \neq u$. Cond. 6 holds, since $\alpha(u)\tilde{c}(u) = \alpha(u)c(u) < \ell(A(u)) + c(u) < \frac{3}{2}\ell(A(u)) < \frac{3}{2}2L = 3L$.

Consider a recursive call in which A' satisfies Cond. 2. That is, $V(A') = \{v_0\}$, $\alpha'(v_0) = 1$, and $\ell(A'(v_0)) \geq c(v_0)/2$. There are three possible options:

$(2 \to 2)$ If $\alpha(u) = 0$ then $A = A'$ and therefore A satisfies Cond. 2.

$(2 \to 6)$ If $\alpha(u) = 1$ then we show that Cond. 6 holds. Consider the charging scheme $\$(v_0) = \$'(v_0) + 2\deg(u)$, $\$(u) = \deg(u)$, and $\$(v) = \$'(v)$ for every $v \neq v_0, u$. Observe that $\deg(u) \leq c(u)$. Hence, $\$(u) = \deg(u) = \alpha(u)\tilde{c}(u)$. Also, $\ell(A(v_0)) < L$ since $\alpha(u) > 0$. Therefore, $\$(v_0) = 3L' + 2\deg(u) \geq 2L \geq 2\ell(A(v_0)) \geq c(v_0)$. Hence, $\$(v) \geq \alpha(v)\tilde{c}(v)$ for every v and Cond. 6 holds.

$(2 \to 4)$ If $\alpha(u) \geq 2$ then we show that Cond. 4 holds. Consider the charging scheme $\$(v_0) = \$'(v_0) + \deg(u)$, $\$(u) = 2\deg(u)$, and $\$(v) = \$'(v)$ for every $v \neq v_0, u$. Since $\ell(A(u)) = \deg(u) > c(u)$ it follows that $\$(u) = 2\deg(u) \geq \alpha(u)c(u)$. Moreover, $\$(v_0) \geq L$ since $\$'(v_0) \geq L'$. Hence, it remains to show that $\$(v_0) \geq 2\ell(A(v_0)) - (\ell(A) - L)$. Since $\alpha(u) > 0$ we know that $\deg(u) > \ell(A') - L'$. Also, observe that $\ell(A') - L' = \ell(A) - L$. Hence, $\$(v_0) \geq 2L' + \deg(u) = 2\ell(A') - 2(\ell(A') - L') + \deg(u) > 2\ell(A(v_0)) - (\ell(A) - L)$.

Consider a recursive call in which A' satisfies Cond. 3. That is, $V(A) = \{v_0\}$, $\alpha(v_0) = 2$, and $\ell(A'(v_0)) - \ell(e_0) < L'$. There are two possibilities:

$(3 \to 3)$ If $\alpha(u) = 0$ then A satisfies Cond. 3, since $\ell(A(v_0)) - \ell(e_0) < L' < L$.

$(3 \to 6)$ If $\alpha(u) > 0$ then we show that Cond. 6 holds. Consider the charging scheme $\$(v_0) = \$'(v_0) + \deg(u)$, $\$(u) = 2\deg(u)$, and $\$(v) = \$'(v)$ for every $v \neq v_0, u$. First, since $\ell(A(u)) = \deg(u) \geq \tilde{c}(u)$ it follows that $\$(u) = 2\deg(u) \geq \alpha(u)\tilde{c}(u)$. As for v_0, first observe that since $\alpha(v_0) = 2$ and $\alpha(u) > 0$ it follows that $c(v_0) < \ell(A(v_0)) < L$. We claim that $A(v_0)$ contains at least two edges, otherwise A' could only have been constructed in Line 2 and we assume the recursion ends in Line 1. Since we add edges to $A(v_0)$ in a non-decreasing order of demands it follows that $\ell(e_0) \leq \ell(A(v_0))/2$. This implies that $L' \geq \ell(A(v_0))/2 > c(v_0)/2$ because $\ell(A'(v_0)) - \ell(e_0) < L'$. It follows that $\$(v_0) = 2L' + L \geq c(v_0) + c(v_0) = \alpha(v_0)\tilde{c}(v_0)$.

The rest of the transitions are omitted for lack of space. □

Lemma 3. *Let A be an assignment that was computed by Algorithm **PCVC** for $G = (V, E)$ and L. Furthermore, assume the recursion ended in Line 1. Then, there exists a charging scheme (with respect to G and assignment A) in which every vertex v is given at least $\alpha(v)b(v)$ coins and $\sum_v \alpha(v)b(v) \leq 3L$.*

Proof. The lemma follows from Lemma 2 and the definition of b. □

4.2 The Recursion Ends with a Medium or Large Edge

In this section we study what happens when Algorithm **PCVC** ends its recursion in Line 2 assigning edge e_x to vertex x. The approach is similar to that used in the previous section, the main difference being that since we want to compare our solution to one that assigns e_x to x we have more coins to distribute. More specifically, we say a charging scheme $\$$ is *valid* if $\sum_v \$(v) \leq 3\max\{L, \alpha(x)c(x)\}$. We show that there is a way to distribute these coins so that $\$(v) \geq \alpha(v)b(v)$. The proof of the next lemma contains a recursive definition of our charging scheme. The proof is omitted for lack of space.

Lemma 4. *Let A be an assignment that was computed by Algorithm **PCVC** for $G = (V, E)$ and L. Furthermore, assume the recursion ended in Line 2 assigning edge e_x to x. Then, one the following conditions must hold:*

1. *$V(A) = \{x\}$. Also, the charging scheme $\$(x) = \alpha(x)c(x)$ and $\$(v) = 0$ for every $v \neq x$ is valid.*
2. *$V(A) = \{x, v_1\}$, $\alpha(v_1) > 1$. Also, the charging scheme $\$(x) = \alpha(x)c(x)$, $\$(v_1) = 2\ell(A(v_1))$ and $\$(v) = 0$ for every $v \neq x, v_1$ is valid.*
3. *$V(A) \neq \emptyset$. Also, there exists a valid charging scheme $\$$ such that $\$(v) \geq \alpha(v)\tilde{c}(v)$ for every $v \in V$ and $\sum_v \$(v) \leq 3L$.*

Lemma 5. *Let A be an assignment that was computed by Algorithm **PCVC** for $G = (V, E)$ and L. Furthermore, assume the recursion ended in Line 2*

assigning edge e_x to x. Then, there exists a charging scheme (with respect G and A) in which every vertex v is given at least $\alpha(v)b(v)$ coins and $\sum_v \alpha(v)b(v) \leq 3\max\{L, \alpha(x)c(x)\}$.

Proof. The lemma follows from Lemma 4 and the definition of b. □

4.3 Proof of Theorem 2

For the sake of brevity when we say that a given assignment is 3-approximate we mean compared to an optimal solution if the recursion of Algorithm **PCVC** ended in Line 1, and compared to the cheapest solution that assigns e_x to x if the recursion ended in Line 2.

Our goal is to prove that the assignment produced by Algorithm **PCVC** is 3-approximate. The proof is by induction on the recursion. In the base case the algorithm returns an empty assignment (if the recursion ends in Line 1) or assigns e_x to x (if the recursion ends in Line 2), in both cases the assignment is optimal. For the inductive step there are three cases.

First, if the recursive call is made in Line 3, then by the inductive hypothesis the assignment A' is 3-approximate with respect to $(V \setminus \{u\}, E)$. A' is clearly 3-approximate with respect to (V, E) since $\deg(u) = 0$.

Second, if the recursive invocation is made in Line 5, then by the inductive hypothesis the assignment A' is 3-approximate with respect to $(V \setminus \{v\}, E \setminus E(u))$, w, and $\max\{L - \deg(u), 0\}$. Since $w(u) = 0$, the optimum with respect to (V, E), w, and L is equal to the optimum with respect to $(V \setminus \{v\}, E \setminus E(u))$, w, and $\max\{L - \deg(u), 0\}$. Moreover, since $\alpha(v) = \alpha'(v)$ for every $v \in V \setminus \{u\}$ due to Obs. 1, it follows that $w(A) = w(A')$. Thus A is 3-approximate with respect to (V, E), w, and L.

Third, if the recursive call is made in Line 14, then by the inductive hypothesis the assignment A' is 3-approximate with respect to (V, E), w_2, and L. If the recursion ended in Line 1 then by Lemma 3 $w_1(A) \leq \varepsilon \cdot 3L$. We show that $w_1(\bar{A}) \geq \varepsilon \cdot L$ for every feasible assignment $\bar{A}$. First, if there exists $v \in V(\bar{A})$ such that $b(v) = L$ then $w_1(\bar{A}) \geq L \cdot \varepsilon$. Otherwise, $b(v) = \tilde{c}(v) < L$ for every $v \in V(\bar{A})$. It follows that $w_1(\bar{A}) = \varepsilon \cdot \sum_v \bar{\alpha}(v)\tilde{c}(v) \geq \varepsilon \cdot \sum_v \ell(\bar{A}(v)) \geq \varepsilon \cdot L$. Hence, if the recursion ended in Line 1, A is 3-approximate with respect to w_1 too, and by the Local Ratio Theorem it is 3-approximation with respect to w as well. On the other hand, if the recursion ended in Line 2 then by Lemma 5 $w_1(A) \leq \varepsilon \cdot 3\max\{L, \alpha(x)c(x)\}$. We need to show that $w_1(\bar{A}) \geq \varepsilon \cdot \max\{L, \alpha(x)c(x)\}$ for any feasible cover $\bar{A}$ that assigns e_x to x. By the previous argument we know that $w_1(\bar{A}) \geq \varepsilon \cdot L$. In addition, because $e \in \bar{A}(x)$, we have $w_1(\bar{A}) \geq \bar{\alpha}(x)w_1(x) \geq \alpha(x)w_1(x) = \varepsilon \cdot \alpha(x)b(x)$. Since the if condition in Line 2 was not met in this call we have $\ell(e_x) \leq L$ and so $b(x) = c(x)$. Thus, if the recursion ended in Line 2, A is 3-approximate with respect to w_1 too, and by the Local Ratio Theorem it is 3-approximate with respect to w as well.

Acknowledgments. We thank Einat Or for helpful discussions. We also thank Roee Engelberg for his suggestions.

References

1. Karp, R.M.: Reducibility among combinatorial problems. Complexity of Computer Computations, 85–103 (1972)
2. Dinur, I., Safra, S.: The importance of being biased. In: 34th ACM Symp. on the Theory of Computing, pp. 33–42 (2002)
3. Hochbaum, D.S. (ed.): Approximation Algorithms for NP-Hard Problem. PWS Publishing Company (1997)
4. Bar-Yehuda, R., Bendel, K., Freund, A., Rawitz, D.: Local ratio: a unified framework for approximation algorithms. ACM Comp. Surveys 36(4), 422–463 (2004)
5. Guha, S., Hassin, R., Khuller, S., Or, E.: Capacitated vertex covering. Journal of Algorithms 48(1), 257–270 (2003)
6. Lustbader, J.W., Puett, D., Ruddon, R.W. (eds.): Symposium on Glycoprotein Hormones: Structure, Function, and Clinical Implications (1993)
7. Wolsey, L.A.: An analysis of the greedy algorithm for the submodular set covering problem. Combinatorica 2, 385–393 (1982)
8. Gandhi, R., Khuller, S., Parthasarathy, S., Srinivasan, A.: Dependent rounding and its applications to approximation algorithms. Journal of the ACM 53(3), 324–360 (2006)
9. Chuzhoy, J., Naor, J.: Covering problems with hard capacities. In: 43nd IEEE Symp. on Foundations of Comp.Sci., pp. 481–489 (2002)
10. Gandhi, R., Halperin, E., Khuller, S., Kortsarz, G., Srinivasan, A.: An improved approximation algorithm for vertex cover with hard capacities. Journal of Computer and System Sciences 72(1), 16–33 (2006)
11. Kearns, M.: The Computational Complexity of Machine Learning. MIT Press, Cambridge (1990)
12. Slavík, P.: Improved performance of the greedy algorithm for partial cover. Information Processing Letters 64(5), 251–254 (1997)
13. Bshouty, N.H., Burroughs, L.: Massaging a linear programming solution to give a 2-approximation for a generalization of the vertex cover problem. In: Meinel, C., Morvan, M. (eds.) STACS 98. LNCS, vol. 1373, pp. 298–308. Springer, Heidelberg (1998)
14. Bar-Yehuda.: Using homogeneous weights for approximating the partial cover problem. Journal of Algorithms 39, 137–144 (2001)
15. Gandhi, R., Khuller, S., Srinivasan, A.: Approximation algorithms for partial covering problems. Journal of Algorithms 53(1), 55–84 (2004)
16. Srinivasan, A.: Distributions on level-sets with applications to approximation algorithms. In: 42nd IEEE Symp. on Foundations of Comp. Sci., pp. 588–597 (2001)
17. Halperin, E., Srinivasan, A.: Improved approximation algorithms for the partial vertex cover problem. In: Jansen, K., Leonardi, S., Vazirani, V.V. (eds.) APPROX 2002. LNCS, vol. 2462, pp. 161–174. Springer, Heidelberg (2002)
18. Mestre, J.: A primal-dual approximation algorithm for partial vertex cover: Making educated guesses. In: Chekuri, C., Jansen, K., Rolim, J.D.P., Trevisan, L. (eds.) APPROX 2005 and RANDOM 2005. LNCS, vol. 3624, pp. 182–191. Springer, Heidelberg (2005)
19. Bar-Yehuda, R., Even, S.: A local-ratio theorem for approximating the weighted vertex cover problem. Annals of Discrete Mathematics 25, 27–46 (1985)
20. Bar-Yehuda, R.: One for the price of two: A unified approach for approximating covering problems. Algorithmica 27(2), 131–144 (2000)

Optimal Resilient Dynamic Dictionaries*

Gerth Stølting Brodal[1], Rolf Fagerberg[2], Irene Finocchi[3], Fabrizio Grandoni[3], Giuseppe F. Italiano[4], Allan Grønlund Jørgensen[1], Gabriel Moruz[1], and Thomas Mølhave[1]

[1] BRICS**, MADALGO***, Department of Computer Science, University of Aarhus, Denmark
{gerth,jallan,gabi,thomasm}@daimi.au.dk
[2] Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
rolf@imada.sdu.dk
[3] Dipartimento di Informatica, Università di Roma "La Sapienza", Italy
{finocchi,grandoni}@di.uniroma1.it
[4] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Italy
italiano@disp.uniroma2.it

Abstract. We investigate the problem of computing in the presence of faults that may arbitrarily (i.e., adversarially) corrupt memory locations. In the faulty memory model, any memory cell can get corrupted at any time, and corrupted cells cannot be distinguished from uncorrupted ones. An upper bound δ on the number of corruptions and $O(1)$ reliable memory cells are provided. In this model, we focus on the design of resilient dictionaries, i.e., dictionaries which are able to operate correctly (at least) on the set of uncorrupted keys. We first present a simple resilient dynamic search tree, based on random sampling, with $O(\log n+\delta)$ expected amortized cost per operation, and $O(n)$ space complexity. We then propose an optimal deterministic static dictionary supporting searches in $\Theta(\log n+\delta)$ time in the worst case, and we show how to use it in a dynamic setting in order to support updates in $O(\log n + \delta)$ amortized time. Our dynamic dictionary also supports range queries in $O(\log n+\delta+t)$ worst case time, where t is the size of the output. Finally, we show that every resilient search tree (with some reasonable properties) must take $\Omega(\log n + \delta)$ worst-case time per search.

* Work partially supported by the Danish Natural Science Foundation (SNF), by the Italian Ministry of University and Research under Project MAINSTREAM "Algorithms for Massive Information Structures and Data Streams", by an Ole Roemer Scholarship from the Danish National Science Research Council, and by a Scholarship from the Oticon Foundation.
** Basic Research in Computer Science, research school.
*** Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 347–358, 2007.

1 Introduction

Memories in modern computing platforms are not always fully reliable, and sometimes the content of a memory word may be temporarily or permanently lost or corrupted. This may depend on manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles [17,22]. Furthermore, latest trends in storage development point out that memory devices are getting smaller and more complex. Additionally, they work at lower voltages and higher frequencies [10]. All these improvements increase the likelihood of soft memory errors, whose rate is expected to increase for both SRAM and DRAM memories [24]. These phenomena can seriously affect the computation, especially if the amount of data to be processed is huge. This is for example the case for Web search engines, which store and process Terabytes of dynamic data sets, including inverted indices which have to be maintained sorted for fast document access. For such large data structures, even a small failure probability can result in bit flips in the index, which may become responsible of erroneous answers to keyword searches [18].

Memory corruptions have been addressed in various ways, both at the hardware and software level. At the hardware level, memory corruptions are tackled using error detection mechanisms, such as redundancy, parity checking or Hamming codes. However, adopting such mechanisms involves non-negligible penalties with respect to performance, size, and cost, thus memories implementing them are rarely found in large scale clusters or ordinary workstations. Dealing with unreliable information has been addressed in the algorithmic community in a variety of different settings, including the liar model [1,5,12,20,23], fault-tolerant sorting networks [2,21,25], resiliency of pointer-based data structures [3], parallel models of computation with faulty memories [9].

A model for memory faults. Finocchi and Italiano [16] introduced the *faulty-memory RAM*. In this model, we assume that there is an adaptive adversary which can corrupt up to δ memory words, in any place and at any time (even simultaneously). We remark that δ is not a constant, but is a parameter of the model. The pessimistic faulty-memory RAM captures situations like cosmic-rays bursts and memories with non-uniform fault-probability, which would be difficult to be modeled otherwise. The model also assumes that there are $O(1)$ safe memory words which cannot be accessed by the adversary. Note that, without this assumption, no reliable computation is possible: in particular, the $O(1)$ safe memory can store the code of the algorithm itself, which otherwise could be corrupted by the adversary. In the case of randomized algorithms, we assume that the random bits are not accessible to the adversary. Moreover, we assume that reading a memory word (in the unsafe memory) is an atomic operation, that is the adversary cannot corrupt a memory word after the reading process has started. Without the last two assumptions, most of the power of randomization would be lost in our setting. An algorithm or a data structure is called *resilient* if it works correctly at least on the set of uncorrupted cells in the input. In particular, a resilient searching algorithm returns a positive answer if there

exists an uncorrupted element in the input equal to the search key. If there is no element, corrupted or uncorrupted, matching the search key, the algorithm returns a negative answer. If there is a corrupted value equal to the search key, the answer can be either positive or negative.

Previous work. Several problems have been addressed in the faulty-memory RAM. In the original paper [16], lower bounds and (non-optimal) algorithms for sorting and searching were given. In particular, it has been proved in [16] that searching in a sorted array takes $\Omega(\log n + \delta)$ time, i.e., up to $O(\log n)$ corruptions can be tolerated while still preserving the classical $O(\log n)$ searching bound. Matching upper bounds for sorting and randomized searching, as well as an $O(\log n + \delta^{1+\varepsilon})$ deterministic searching algorithm, were then given in [14]. Resilient search trees that support searches, insertions, and deletions in $O(\log n + \delta^2)$ amortized time were introduced in [15]. Recently, Jørgensen *et al.* [19] proposed priority queues supporting both insert and delete-min operations in $O(\log n + \delta)$ amortized time. Finally, in [13] it was empirically shown that resilient sorting algorithms are of practical interest.

Our contribution. In this paper we continue the work on resilient dictionaries. We present a simple randomized dynamic search tree achieving $O(\log n + \delta)$ amortized expected time per operation. We then present the first resilient algorithm for deterministically searching in a sorted array in optimal $O(\log n + \delta)$ time, matching the lower bound from [16]. We use this algorithm, to build a resilient deterministic dynamic dictionary supporting searches in $O(\log n + \delta)$ worst case time and updates in $O(\log n + \delta)$ amortized time. Range queries are supported in $O(\log n + \delta + t)$ time where t is the size of the output. Furthermore, we prove a lower bound stating that every resilient dictionary (with some reasonable properties) must take $\Omega(\log n + \delta)$ worst-case time per search.

Preliminaries. We denote by α the *actual* number of faults. Of course $\alpha \leq \delta$. A *resilient variable* x consists of $(2\delta + 1)$ copies of a (classical) variable. The *value* of x is the majority value of its copies. This value is well defined since at most δ copies can be corrupted. Assigning a value to x means assigning such value to all the copies of x. Note that both reading and updating x can be done in $O(\delta)$ time and $O(1)$ space (using, e.g., the algorithm for majority computation in [6]).

2 A Simple Randomized Resilient Dictionary

In this section we present a simple randomized resilient search tree, which builds upon the resilient search tree of [15]. Our search tree takes $O(\log n+\delta)$ amortized time per operation, in expectation. We maintain a dynamically evolving set of intervals $I_1, I_2, \ldots I_h$. Initially, when the set of keys is empty, there is a unique interval $I_1 = (-\infty, +\infty)$. Throughout the sequence of operations we maintain the following invariants:

(i) The intervals are non-overlapping, and their union is $(-\infty, +\infty)$.
(ii) Each interval contains less than 2δ keys.

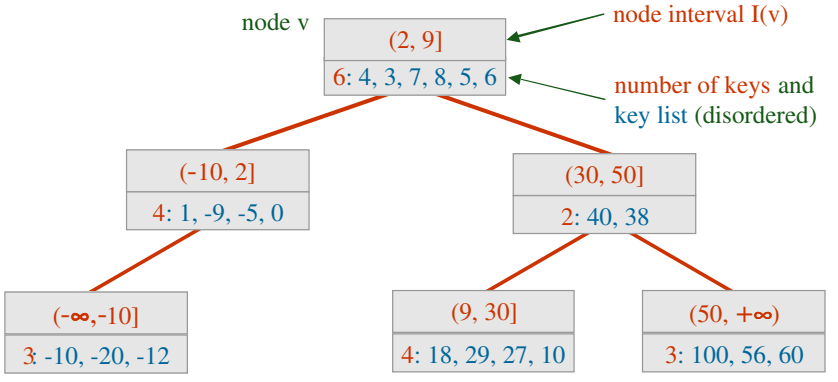


Fig. 1. A resilient search tree

(iii) Each interval contains more than $\delta/2$ keys, except possibly for the leftmost and the rightmost intervals (*boundary* intervals).

To implement any search, insert, or delete of a key e, we first need to find the interval $I(e)$ containing e (*interval search*). Invariant (i) guarantees that such an interval exists, and is unique. Invariants (ii) and (iii) have a crucial role in the amortized analysis of the algorithm, as we will clarify later.

The intervals are maintained in a standard balanced binary search tree. Throughout the paper we use as a reference implementation an *AVL* tree [11]. However, the same basic approach also works with other search trees. Intervals are ordered according to, say, their left endpoints. For each node v of the search tree, we maintain the following variables:

1. (reliably) the endpoints of the corresponding interval $I(v)$ and the number $|I(v)|$ of keys contained in the interval $I(v)$;
2. (reliably) the addresses of the left child, the right child, and the parent of v, and all the information needed to keep the search tree balanced with the implementation considered;
3. (unreliably, i.e., in a single copy) the (unordered) set of current keys contained in $I(v)$, stored in an array of size 2δ.

For an example, see Figure 1. The nodes of the search tree are stored in an array. The main reason for this is that it makes it easy to check whether a pointer/index points to a search tree node. Otherwise, the algorithm could jump outside of the search tree by following a corrupted pointer, without even noticing it: this would make the behavior of the algorithm unpredictable. The address of the array and the current number of nodes is kept in safe memory, together with the address of the root node. We use a standard *doubling* technique to ensure that the size of the array is linear in the current number of nodes. The amortized overhead per insertion/deletion of a node due to doubling is $O(\delta)$. As we will see, this cost can be charged to the cost of the interval search which is performed in each operation: from now on we will not mention this cost any further.

We next describe how to search, insert, and delete a given key e.

Search. Every search is performed by first searching for the interval $I(e)$ containing e (*interval search*), and then by linearly searching for e in $I(e)$. The interval search is implemented as follows. We perform a classical search, where for each relevant resilient variable we consider one of its $2\delta + 1$ copies chosen uniformly at random. We implement the search so as to read at most one random copy of each resilient variable (unless pointers corruption forces the search to cycle). This assumption will turn out to be useful in the analysis. Once the search is concluded, we check reliably (in $O(\delta)$ time) whether the final interval contains e. If not, the search is restarted from scratch. The search is restarted also as soon as an inconsistency is found, for instance if the number of steps performed gets larger than the height of the tree.

Insert. We initially find $I = I(e)$ with the procedure above. Then, if e is not already in the list of keys associated to I, we add e to such list. If the size of the list becomes 2δ because of this insertion, we perform the following operations in order to preserve Invariant (ii). We delete interval I from the search tree, and we split I in two non-overlapping subintervals L and R, $L \cup R = I$, which take the smaller and larger half of the keys of I, respectively. In order to split the keys of I in two halves, we use two-way BubbleSort as described in [14]. This takes time $O(\delta^2)$. Eventually, we insert L and R in the search tree. Both deletion and insertion of intervals from/in the search tree are performed in the standard way (with rotations for balancing), but using resilient variables only, hence in time $O(\delta \log n)$. Note that Invariants (i) and (iii) are preserved.

Delete. We first find $I = I(e)$ using the search procedure above. If we find e in the list of keys associated to I, we delete e. Then, if $|I| = \delta/2$ and I is not a boundary interval, we perform, reliably, the following operations in order to preserve Invariant (iii). First, we search the interval L to the left of I, and delete both L and I from the search tree. Then we do two different things, depending on the size of L. If $|L| \leq \delta$, we merge L and I into a unique interval $I' = L \cup I$, and insert I' in the search tree. Otherwise ($|L| > \delta$), we create two new non-overlapping intervals L' and I' such that $L' \cup I' = L \cup I$, L' contains all the keys of L but the $\delta/4$ largest ones, and I' contains the remaining keys of $L \cup I$. Also in this case creating L' and I' takes time $O(\delta^2)$ with two-way BubbleSort. We next insert intervals L' and I' into the search tree. Again, the cost per insertion/deletion of an interval is $O(\delta \log n)$, since we use resilient variables. Observe that Invariants (i) and (ii) are preserved.

By Invariant (iii), the total number of nodes in the search tree is $O(1 + n/\delta)$. Since each node takes $\Theta(\delta)$ space, and thanks to doubling, the space occupied by the search tree is $O(n + \delta)$. This is also an upper bound on the overall space complexity. The space complexity can be reduced to $O(n)$ by storing the variables associated to boundary intervals in the $O(1)$ size safe memory, and by handling the corresponding set of keys via doubling. This change can be done without

affecting the running time of the operations. We remark that the implementation of the interval search is the main difference between our improved search tree and the search tree in [15].

Theorem 1. *The resilient search tree above has $O(\log n+\delta)$ expected amortized time per operation and $O(n)$ space complexity.*

Proof. The space complexity is discussed above. Let $S(n,\delta)$ be the expected time needed to perform an interval search. By Invariant (ii), each search operation takes $S(n,\delta)+O(\delta)$ expected time. The same holds for insert and delete operations, when the structure of the search tree has not to be modified. Otherwise, there is an extra $O(\delta\log n+\delta^2)$ cost. However, it is not hard to show that, by Invariants (ii) and (iii), the search tree is modified every $\Omega(\delta)$ insert and/or delete operations (see [15] for a formal proof of this simple fact). Hence the amortized cost of insert and delete operations is $S(n,\delta)+O(\log n+\delta)$ in expectation.

It remains to bound $S(n,\delta)$. Each search round takes $O(\log n+\delta)$ time. Thus it is sufficient to show that the expected number of rounds is constant. Consider a given round. Let α_i be the actual number of faults happening at any time on the i-th resilient variable considered during the round, $i=1,2,\ldots,p$. The probability that all the copies chosen during a given round are faithful is at least

$$\left(1-\frac{\alpha_1}{2\delta+1}\right)\left(1-\frac{\alpha_2}{2\delta+1}\right)\cdots\left(1-\frac{\alpha_p}{2\delta+1}\right)\geq\left(1-\frac{\sum_{i=1}^{p}\alpha_i}{2\delta+1}\right).$$

Given this event, by the assumptions on the algorithm the resilient variables considered must be all distinct. As a consequence $\sum_{i=1}^{p}\alpha_i\leq\alpha\leq\delta$, and hence

$$\left(1-\frac{\sum_{i=1}^{p}\alpha_i}{2\delta+1}\right)\geq\left(1-\frac{\delta}{2\delta+1}\right)\geq\frac{1}{2}.$$

It follows that the expected number of rounds is at most 2. □

3 An Optimal Static Dictionary

In this section we close the gap between lower and upper bounds for deterministic resilient searching algorithms. We present a resilient algorithm that searches for an element in a sorted array in $O(\log n+\delta)$ time in the worst case, which is optimal [16]. The previously best known deterministic dictionary supports searches in $O(\log n+\delta^{1+\varepsilon})$ time [14].

We design a binary search algorithm, which only advance one level in the wrong direction for each corrupted element misleading it. We then design a verification procedure that checks the result of the binary search. We count the number of detected corruptions and adjust our algorithm accordingly to ensure that no element is used more than once. To avoid reading the same faulty value twice, we divide the input array into implicit blocks. Each block consists of $5\delta+1$ consecutive elements of the input and is structured in three segments: the *left*

verification segment, LV, consists of the first 2δ elements, the next $\delta+1$ elements form the *query segment*, Q, and the *right verification segment*, RV, consists of the last 2δ elements of the block. The left and right verification segments, LV and RV, are used only by the verification procedure. The elements in the query segment are used to define $\delta + 1$ sorted sequences $S_0, \ldots, S_\delta$. The j'th element of sequence S_i, $S_i[j]$, is the i'th element of the query segment of the j'th block, and is located at position $\text{pos}_i(j) = (5\delta + 1)j + 2\delta + i$ in the input array.

We store a value $k \in \{0, \ldots, \delta\}$ in safe memory identifying the sequence S_k on which we perform the binary search. Also, k identifies the number of corruptions detected. Whenever we detect a corruption, we change the sequence on which we perform the search by incrementing k. Since there are $\delta + 1$ disjoint sequences, there exists at least one sequence without any corruptions.

Binary search. The binary search is performed on the elements of S_k. We store in safe memory the search key, e, and the left and right sequence indices, l and r, used by the binary search. Initially, $l = -1$ is the position of an implicit $-\infty$ element. Similarly, r is the position of an implicit ∞ to the right of the last element. Since each element in S_k belongs to a distinct block, l and r also identify two blocks B_l and B_r.

In each step of the binary search the element at position $i = \lfloor (l + r)/2 \rfloor$ in S_k is compared with e. Assume without loss of generality that this element is smaller than e. We set l to i and decrement r by one. We then compare e with $S_k[r]$. If this element is larger than e, the search continues. Otherwise, if no corruptions have occurred, the position of the search element is in block B_r or B_{r+1} in the input array. When two adjacent elements are identified as in the case just described, or when l and r become adjacent, we invoke a verification procedure on the corresponding blocks.

The verification procedure determines whether the two adjacent blocks, denoted B_i and B_{i+1}, are correctly identified. If the verification succeeds, the binary search is completed, and all the elements in the two corresponding adjacent blocks, B_i and B_{i+1} are scanned. The search returns true if e is found during the scan, and false otherwise. If the verification fails, we backtrack the search two steps, since it may have been mislead by corruptions. To facilitate backtracking, we store two word-sized bit-vectors, d and f in safe memory. The i'th bit of d indicates the direction of the search and the i'th bit of f indicates whether there was a rounding in computing the middle element in the i'th step of the binary search respectively. We can compute the values of l and r in the previous step by retrieving the relevant bits of d and f. If the verification fails, it detects at least one corruption and therefore k is incremented, thus the search continues on a different sequence S_k.

Verification phase. Verification is performed on two adjacent blocks, B_i and B_{i+1}. It either determines that e lies in B_i or B_{i+1} or detects corruptions. The verification is an iterative algorithm maintaining a value which expresses the confidence that the search key resides in B_i or B_{i+1}. We compute the *left confidence*, c_l, which is a value that quantifies the confidence that e is in B_i or to

the right of it. Intuitively, an element in LV_i smaller than e is consistent with the thesis that e is in B_i or to the right of it. However, an element in LV_i larger than e is inconsistent. Similarly, we compute the *right confidence*, c_r, to express the confidence that e is in B_{i+1} or to the left of it.

We compute c_l by scanning a sub-interval of the left verification segment, LV_i, of B_i. Similarly, the right confidence is computed by scanning the right verification segment, RV_{i+1}, of B_{i+1}. Initially, we set $c_l = 1$ and $c_r = 1$. We scan LV_i from right to left starting at the element at index $v_l = 2\delta - 2k$ in LV_i. Similarly, we scan RV_{i+1} from left to right beginning with the element at position $v_r = 2k$. In an iteration we compare $LV_i[v_l]$ and $RV_{i+1}[v_r]$ against e. If $LV_i[v_l] \leq e$, we increment c_l by one, otherwise it is decreased by one and k is increased by one. Similarly, if $RV_{i+1}[v_r] \geq e$, we increment c_r by one; otherwise, we decrease c_r and increase k. The verification procedure stops when $\min(c_r, c_l)$ equals $\delta - k + 1$ or 0. The verification succeeds in the former case, and fails in the latter.

Theorem 2. *The algorithm is resilient and searches for an element in a sorted array in $O(\log n + \delta)$ time.*

Proof. We first prove that when c_l or c_r decrease during verification, a corruption has been detected. We increase c_l when an element smaller than e is encountered in LV_i, and decrease it otherwise. Intuitively, c_l can been seen as the size of a stack S. When we encounter an element smaller than e, we treat it as if it was pushed, and as if a pop occurred otherwise. Initially, the element g from the query segment of B_i used by the binary search is pushed in S. Since g was used to define the left boundary in the binary search, $g < e$ at that time. Each time an element $LV_i[v] < e$ is popped from the stack, it is *matched* with the current element $LV_i[v_l]$. Since $LV_i[v] < e < LV_i[v_l]$ and $v_l < v$, at least one of $LV_i[v_l]$ and $LV_i[v]$ is corrupted, and therefore each match corresponds to detecting at least one corruption. It follows that if $2t - 1$ elements are scanned on either side during a failed verification, then at least t corruptions are detected.

We now argue that no single corruption is counted twice. A corruption is detected if and only if two elements are matched during verification. Thus it suffices to argue that no element participates in more than one matching. We first analyze corruptions occurring in the left and right verification segments. Since the verification starts at index $2(\delta - k)$ in the left verification segment and k is increased when a corruption is detected, no element is accessed twice, and therefore not matched twice either. A similar argument holds for the right verification segment. Each failed verification increments k, thus no element from a query segment is read more than once. In each step of the binary search both the left and the right indices are updated. Whenever we backtrack the binary search, the last two updates of l and r are reverted. Therefore, if the same block is used in a subsequent verification, a new element from the query segment is read, and this new element is the one initially on the stack. We conclude that elements in the query segments, which are initially placed on the stack, are never matched twice either.

To argue correctness we prove that if a verification is successful, and e is not found in the scan of the two blocks, then no uncorrupted element equal to e

exists in the input. If a verification succeeded then $c_l \geq \delta - k + 1$. Since only $\delta - k$ more corruptions are possible and since an element equal to e was not found, there is at least one uncorrupted element in LV_i smaller than e, and thus there can not be any uncorrupted elements equal to e to the left of B_i in the input array. By a similar argument, if $c_r \geq \delta - k + 1$, then all uncorrupted elements to the right of B_{i+1} in the input array are larger than e.

We now analyze the running time. We charge each backtracking of the binary search to the verification procedure that triggered it. Therefore, the total time of the algorithm is $O(\log n)$ plus the time required by verifications. To bound the time used for all verification steps we use the fact that if $O(f)$ time is used for a verification step, then $\Omega(f)$ corruptions are detected or the algorithm ends. At most $O(\delta)$ time is used in the last verification for scanning the two blocks. □

4 A Dynamic Dictionary

In this section we describe a linear space resilient dynamic dictionary supporting searches in optimal $O(\log n + \delta)$ worst case time and range queries in optimal $O(\log n + \delta + t)$ worst case time, where t is the size of the output. The amortized update cost is $O(\log n + \delta)$. The previous best known deterministic dynamic dictionary, is the resilient search tree of [15], which supports searches and updates in $O(\log n + \delta^2)$ amortized time.

Structure. The sorted sequence of elements is partitioned into a sequence of *leaf structures*, each storing $\Theta(\delta \log n)$ elements. For each leaf structure we select a guiding element, and these $O(n/(\delta \log n))$ elements are also stored in the leaves of a reliably stored binary search tree. Each guiding element is larger than all uncorrupted elements in the corresponding leaf structure.

For this reliable *top tree* T, we use the binary search tree in [7], which consists of $h = \log |T| + O(1)$ levels when containing $|T|$ elements. In the full version [8] it is shown how the tree can be maintained such that the first $h - 2$ levels are complete. We lay out the tree in memory in left-to-right breadth first order, as specified in [7]. It uses linear space, and supports updates in $O(\log^2 |T|)$ amortized time. Global rebuilding is used when $|T|$ changes by a constant factor.

All the elements in the top tree are stored as resilient variables. . Since a resilient variable takes $O(\delta)$ space, $O(\delta|T|)$ space is used for the entire structure. The time used for storing and retrieving a resilient variable is $O(\delta)$, and therefore the additional work required to handle the resilient variables increases the amortized update cost to $O(\delta \log^2 |T|)$ time.

The leaf structure consists of a top bucket B and b buckets, $B_0, \ldots, B_{b-1}$, where $\log n \leq b \leq 4 \log n$. Each bucket B_i contains between δ and 6δ input elements, stored consecutively in an array of size 6δ, and uncorrupted elements in B_i are smaller than uncorrupted elements in B_{i+1}. For each bucket B_i, the top bucket B associates a guiding element larger than all elements in B_i, a pointer to B_i, and the size of B_i, all stored reliably. Since storing a value reliably uses $O(\delta)$ space, the total space used by the top bucket is $O(\delta \log n)$. The guiding elements of B are stored as a sorted array to enable fast searches.

Searching. The search operation first finds a leaf in the top tree, and then searches the corresponding leaf structure. Let h denote the height of T. If $h \leq 3$, we perform a standard tree search from the root of T using the reliably stored guiding elements. Otherwise, we locate internal nodes, v_1 and v_2, with guiding elements g_1 and g_2, such that $g_1 < e \leq g_2$, where e is the search key. Since $h-2$ is the last complete level of T, level $\ell = h-3$ is complete and contains only internal nodes. The breadth first layout of T ensures that elements of level ℓ are stored consecutively in memory. The search operation locates v_1 and v_2 using the deterministic resilient search algorithm from Section 3 on the array defined by level ℓ. The search only considers the $2\delta+1$ cells in each node containing guiding elements and ignores memory used for auxiliary information. Although they are stored using as resilient variables, each of the $2\delta+1$ copies are considered a single element by the search. We modify the resilient searching algorithm previously introduced such that it reports two consecutive blocks with the property that if the search key is in the structure, it is contained in one of them. The reported two blocks, each of size $5\delta+1$, span $O(1)$ nodes of level ℓ and the guiding elements of these are queried reliably to locate v_1 and v_2. The appropriate leaf can be in either of the subtrees rooted at v_1 and v_2, and we perform a standard tree search in both using the reliably stored guiding elements. Searching for an element in a leaf structure is performed by using the resilient search algorithm from Section 3 on the top bucket, B, similar to the way v_1 and v_2 were found in T. The corresponding reliably stored pointer is then followed to a bucket B_i, which is scanned. Range queries can be performed by scanning the level ℓ, starting at v, and reporting relevant elements in the leaves below it.

Updates. Efficiently updating the structure is performed using standard bucketing techniques. To insert an element into the dictionary, we first perform a search to locate the appropriate bucket B_i in a leaf structure, and then the element is appended to B_i and the size updated. When the size of B_i increases to 6δ, we split it into two buckets. We compute a guiding element that splits B_i in $O(\delta^2)$ time by repeatedly scanning B_i and extracting the minimum element. The element m returned by the last iteration is kept in safe memory. In each iteration, we select a new m which is the minimum element in B_i larger than the current m. Since at most δ corruptions can occur, B_i contains at least 2δ uncorrupted elements smaller than m and 2δ uncorrupted elements larger, after $|B_i|/2$ iterations. The new split element is reliably inserted in the top bucket using an insertion sort step in $O(\delta \log n)$ time. Similarly, when the degree the top bucket becomes $4 \log n$, it is split in two new leaf structures in $O(\delta \log n)$ time, and a new guiding element is inserted into the top tree. Deletions are handled similarly.

Theorem 3. *The resilient dynamic dictionary structure uses $O(n)$ space while supporting searches in $O(\log n + \delta)$ time worst case with an amortized update cost of $O(\log n + \delta)$. Range queries with an output size of t is performed in worst case $O(\log n + \delta + t)$ time.*

5 A Lower Bound

In the following we restrict our attention to comparison based dictionaries where the keys are stored in one or more arrays, and the address of each array is maintained in one or more pointers. These assumptions can be partially relaxed. However, we do not discuss such relaxations since they do not add much to the discussion below.

Theorem 4. *Every resilient search tree of the kind above requires* $\Omega(\log n + \delta)$ *worst-case time per search.*

Proof. Every search tree, even in a system without memory faults, takes $\Omega(\log n)$ worst-case time per search. This lower bound extends immediately to the case of resilient search trees. Hence, without loss of generality, assume that $\log n = o(\delta)$. Under this assumption, it is sufficient to show that the time required by a resilient search operation is $\Omega(\delta)$.

Consider any given search tree ST of the kind considered. Let $K_1, K_2, \ldots, K_p$ be the arrays in unsafe memory where (subset of) the keys are maintained. For each K_i there must be at least one pointer containing its address. Recall that only a constant number of keys can be kept in safe memory. Hence $\Theta(n)$ keys are stored in unsafe memory only.

Suppose there is an array K_i containing $\Omega(n)$ keys. Then, by the lower bound on static resilient searching [14], searching for a given key in K_i takes time $\Omega(\log n + \delta) = \Omega(\delta)$. Hence, let us assume that all the arrays contain $o(n)$ keys. Since there are $\omega(1)$ arrays, not all the corresponding pointers can be kept in safe memory. In particular, there must be an array K_i whose pointers are all maintained in the unsafe memory. Assume that we search for a faithful key e contained in K_i. Suppose by contradiction that ST concludes the search in $o(\delta)$ time. This means that ST can read only $o(\delta)$ pointers to reach K_i, and at most $o(\delta)$ keys of K_i. Then an adversary, by corrupting $o(\delta)$ memory words only, can make ST answer no to the query. This contradicts the resiliency of ST, which in the case considered must always find a key equal to e. □

References

1. Aslam, J.A., Dhagat, A.: Searching in the presence of linearly bounded errors. In: ACM STOC'91, pp. 486–493
2. Assaf, S., Upfal, E.: Fault-tolerant sorting networks. SIAM J. Discrete Math. 4(4), 472–480 (1991)
3. Aumann, Y., Bender, M.A.: Fault-tolerant data structures. In: IEEE FOCS'96, pp. 580–589
4. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: IEEE FOCS'91
5. Borgstrom, R.S., Rao Kosaraju, S.: Comparison based search in the presence of errors. In: ACM STOC'93, pp. 130–136.
6. Boyer, R., Moore., S.: MJRTY: - A fast majority vote algorithm. University of Texas Tech. Report (1982)

7. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache-oblivious search trees via binary trees of small height. ACM-SIAM SODA'02, 39–48
8. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache-oblivious search trees via trees of small height. Technical Report ALCOMFT-TR-02-53, ALCOM-FT (May 2002)
9. Chlebus, B.S., Gambin, A., Indyk, P.: Shared-memory simulations on a faulty-memory DMM. In: ICALP'96, pp. 586–597
10. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. IEEE micro 23(4), 14–19 (2003)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press/McGraw-Hill Book Company (2001)
12. Feige, U., Raghavan, P., Peleg, D., Upfal, E.: Computing with noisy information. SIAM Journal on Computing 23, 1001–1018 (1994)
13. Petrillo, U.F., Finocchi, I., Italiano, G.F.: The price of resiliency: a case study on sorting with memory faults. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 768–779. Springer, Heidelberg (2006)
14. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal sorting and searching in the presence of memory faults. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 286–298. Springer, Heidelberg (2006)
15. Finocchi, I., Grandoni, F., Italiano, G.: Resilient search trees. In: ACM-SIAM SODA'07, pp. 547–555.
16. Finocchi, I., Italiano, G.F.: Sorting and searching in faulty memories. Algorithmica ACM STOC'04, pp. 101–110 (Extended abstract) (to appear)
17. Hamdioui, S., Al-Ars, Z., de Goor, J.V., Rodgers, M.: Dynamic faults in random-access-memories: Concept, faults models and tests. Journal of Electronic Testing: Theory and Applications 19, 195–205 (2003)
18. Henzinger, M.R.: Combinatorial algorithms for web search engines - three success stories. In: ACM-SIAM SODA'07 (invited talk)
19. Jørgensen, A.G., Moruz, G., Mølhave, T.: Priority queues resilient to memory faults. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, Springer, Heidelberg (2007)
20. Kleitman, D.J., Meyer, A.R., Rivest, R.L., Spencer, J., Winklmann, K.: Coping with errors in binary search procedures. Journal of Computer and System Sciences 20, 396–404 (1980)
21. Leighton, T., Ma, Y.: Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. SIAM Journal on Computing 29(1), 258–273 (1999)
22. May, T.C., Woods, M.H.: Alpha-particle-induced soft errors in dynamic memories. IEEE Transactions on Electron Devices 26(2) (1979)
23. Pelc, A.: Searching games with errors: Fifty years of coping with liars. Theoretical Computer Science 270, 71–109 (2002)
24. Tezzaron Semiconductor. Soft errors in electronic memory - a white paper (2004), http://www.tezzaron.com/about/papers/papers.html
25. Yao, A.C., Yao, F.F.: On fault-tolerant networks for sorting. SIAM Journal on Computing 14, 120–128 (1985)

Determining the Smallest k Such That G Is k-Outerplanar

Frank Kammer

Institut für Informatik, Universität Augsburg, 86135 Augsburg, Germany
kammer@informatik.uni-augsburg.de

Abstract. The outerplanarity index of a planar graph G is the smallest k such that G has a k-outerplanar embedding. We show how to compute the outerplanarity index of an n-vertex planar graph in $O(n^2)$ time, improving the previous best bound of $O(k^3n^2)$. Using simple variations of the computation we can determine the radius of a planar graph in $O(n^2)$ time and its depth in $O(n^3)$ time.

We also give a linear-time 4-approximation algorithm for the outerplanarity index and show how it can be used to solve maximum independent set and several other NP-hard problems faster on planar graphs with outerplanarity index within a constant factor of their treewidth.

Keywords: Outerplanarity index, k-outerplanar, fixed-parameter algorithms, NP-hard, SPQR trees.

1 Introduction

A promising approach to solve NP-hard graph problems to optimality is to deal with fixed-parameter algorithms, i.e., the aim is to solve NP-hard problems on an n-vertex graph in $O(f(k) \cdot n^{O(1)})$ time for some function f that depends only on some parameter k but not on n. For example, we can use the so-called *treewidth* as the parameter k. The treewidth of a graph G measures the minimum width of a so-called *tree decomposition* of G and, intuitively, describes how treelike G is. Tree decompositions and treewidth were introduced by Robertson and Seymour [15]. Given a tree decomposition of width k of an n-vertex graph G, one can solve many NP-hard problems such as maximum 3-coloring, maximum independent set, maximum triangle matching, minimum edge dominating set, minimum dominating set, minimum maximal matching and minimum vertex cover on G in $O(c^k n)$ time for some constant c. For a definition of these problems see, e.g., [1] and [8].

Many algorithms for computing the treewidth have been published. Reed [13] showed that a tree decomposition of width $O(k)$ can be found for an n-vertex graph G of treewidth k in $O(3^{3k}k \cdot n \log n)$ time. Thus, many NP-hard problems can be solved on G in $O(c^k n \log n)$ time for a constant c. Moreover, Bodlaender [4] gave an algorithm that computes a tree decomposition of width k of an n-vertex graph G of treewidth k in $\Theta(f(k)n)$ time for some exponential function f. The author only states that $f(k)$ is very large; however, Röhrig [14] shows that

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 359–370, 2007.

$f = 2^{\Theta(k^3)}$. Much research for finding tree decompositions has also considered special classes of graphs, e.g., the (k-outer)planar graphs.

Definition 1. *An embedding φ of a planar graph is* 1*-outerplanar if it is outerplanar, i.e., all vertices are incident on the outer face in φ. An embedding of a planar graph is k-outerplanar if removing all vertices on the outer face (together with their incident edges) yields a $(k-1)$-outerplanar embedding.*

A graph is k-outerplanar *if it has a k-outerplanar embedding. The* outerplanarity index *of a graph G is the smallest k such that G is k-outerplanar.*

Using the ratcatcher algorithm of Seymour and Thomas [17] and the results of Gu and Tamaki [9], one can obtain a tree decomposition of width $O(k)$ for a k-outerplanar graph in $O(n^3)$ time. All of the NP-hard problems mentioned above remain NP-hard even on planar graphs [8]. In the special case of planar graphs, the outerplanarity index is a very natural parameter to use for a fixed-parameter algorithm. Bienstock and Monma [2] showed that for a planar n-vertex graph, the outerplanarity index k and a k-outerplanar embedding can be found in $O(k^3n^2)$ time. A general technique due to Baker [1] enables us to solve each of the NP-hard problems mentioned above on k-outerplanar n-vertex graphs G in $O(c^k n)$ time for a constant c if a k-outerplanar embedding of G is given.

Using a new and simple algorithm presented here, we can find the outerplanarity index k and a k-outerplanar embedding in $O(n^2)$ time. Moreover, a slightly modified version of the new algorithm is 4-approximative and runs in linear time. Using the approximation algorithm and Baker's technique, we can solve many NP-hard problems to optimality on k-outerplanar graphs in $O(c^k n)$ time for some constant c (e.g., maximum independent set in $O(8^{4k}n)$ time and maximum triangle matching in $O(16^{4k}n)$ time). Thus, this approach is the fastest for many NP-hard problems for planar graphs whose outerplanarity index k is within a constant factor of their treewidth. Moreover, given a k-outerplanar graph, using the new algorithms one can find tree decompositions of width $3k-1$ and $12k-4$ in $O(n^2)$ time and in $O(kn)$ time, respectively [5,16].

In the following we will consider only the outerplanarity index. However our approach can also be used to determine other distance measures such as the radius and the width. For a definition of these distance measures, see [2].

2 Ideas of the Algorithm

An often used technique is to remove some subgraph C from a given graph G and solve a problem recursively on the remaining graph G'. (G' is later called the *induced graph* of G and C.) Unfortunately, the outerplanarity index of G is not a function of the outerplanarity indices of G' and C alone. However, one can determine the outerplanarity index of G by a computation of the so-called *weighted outerplanarity index* of C—this is a small generalization of the outerplanarity index—that additionally takes into account the weighted outerplanarity indices of $\ell \in \mathbb{N}$ different graphs, each of which is G' with a few additional edges. It turns out that by a recursive call one can compute the weighted outerplanarity

indices of the ℓ graphs simultaneously. The exact value of ℓ depends on the distance measure. For the (weighted) outerplanarity index and the radius, ℓ is 6, and for the width, ℓ is the number of edges. One can observe that this leads to time bounds of $O(n^2)$ for determining the radius of an n-vertex planar graph and $O(n^3)$ time for determining its depth.

As we observe later, the computation of the weighted outerplanarity index is a slight modification of the computation of the outerplanarity index. For the time being let us consider only the outerplanarity index. For a fast and simple computation of the outerplanarity index of C, we have to choose C in a special way. Before going into the details, we need additional terminology. A *rooted embedding* of a planar graph G is a combinatorial embedding of G with a specified outer face. An *embedded graph* (G, φ) is a planar graph G with a rooted embedding φ. A graph G is *biconnected* (*triconnected*) if no removal of one vertex (two vertices) from G disconnects G. As an auxiliary tool for describing the computation of the outerplanarity index, we consider so-called *peelings*. These are defined precisely in the next section. For now it will suffice to think of a peeling as a process that removes the vertices of a graph in successive steps, each of which removes all vertices incident on the outer face. Define the *peeling index* of an embedded graph (G, φ) as the minimal number of steps to remove all vertices of G. Let us call an embedding *optimal* (*c-approximative*) if it is a rooted embedding φ of a planar graph G such that the peeling index of (G, φ) equals (is bounded by c times) the outerplanarity index of G.

Suppose that C is a triconnected graph. A theorem of Whitney [19] states that the combinatorial embedding of C is unique. Moreover, it can be found in linear time [3,12]. Hence, if one face f is chosen as the outer face, the rooted embedding φ of C is also unique. Obviously, we can determine the peeling index k of (C, φ) in linear time. Let φ^{OPT} be an optimal embedding of C. Define f^{OPT} as the outer face of φ^{OPT} and k^{OPT} as the peeling index of $(C, \varphi^{\mathrm{OPT}})$. If the *distance* of two faces f_1 and f_2 is taken to be the minimal number of edges we have to cross by going from f_1 to f_2, the distance in φ^{OPT} from f to f^{OPT} and from f^{OPT} to any other face is at most k^{OPT}. Consequently, k is at most $2k^{\mathrm{OPT}}$. By iterating over all ($\leq 2n$) faces of a combinatorial embedding of G, we find an optimal embedding. Therefore we can conclude the following.

Corollary 2. *Given a triconnected graph G, a 2-approximative and an optimal embedding of G can be obtained in linear and in quadratic time, respectively.*

3 Peelings and Induced Graphs

A *separation vertex* (*separation pair*) of a graph G is a vertex (a pair of vertices) whose removal from G disconnects G. A *biconnected component* of G is a maximal biconnected subgraph of G. Say (G, φ) is biconnected and triconnected, respectively, if G is. Call a function a *weight function* for G if it maps each vertex and each edge of G to a non-negative rational number. Let us call a vertex or edge *outside* in a rooted embedding φ if it is incident on the outer face of φ. For an edge e in an embedded graph (G, φ), define $\varphi - e$ as the embedding obtained

by removing e and merging the two faces f_1 and f_2 incident on e in φ to a face f_3 in $\varphi - e$. Then f_3 *contains* the faces f_1 and f_2. Let $G = (V_G, E_G)$ be a connected embedded graph with weight function r and let φ be a rooted embedding of G. Define $Out(\varphi)$ as the set of outside vertices. If $\mathcal{S} = Out(\varphi)$ and $u, v \in \mathcal{S}$, $\mathcal{S}^{\varphi}_{u \to v} \subset \mathcal{S}$ is the set of vertices visited on a shortest clockwise travel around the outer face of φ from u to v, including u and v. If $u = v$, $\mathcal{S}^{\varphi}_{u \to v} = \{u\}$. Given a directed edge (u, v), we say that an embedding φ' is obtained from φ by *adding* (u, v) *clockwise if* φ' is a (planar) rooted embedding of $(V_G, E_G \cup \{\{u, v\}\})$ such that all inner faces of φ are also inner faces of φ' and $Out(\varphi') = Out(\varphi)^{\varphi}_{v \to u}$. By iterating over a set of directed edges E^+ we can *add* E^+ *clockwise* into an embedded graph. The embedding obtained does not depend on the order in which edges are added since the edges in E^+ can be added clockwise only if they do not interlace. Let E^+ be a set of directed edges that all can be added clockwise into φ. Call the resulting embedding φ^+. The *peeling* $\mathcal{P}$ of the quadruple (G, φ, r, E^+) is the (unique) list of vertex sets $(V_1, \ldots, V_t)$ with $\bigcup_{i=1}^{t} V_i = V_G$ and $V_t \neq \emptyset$ such that each set V_i $(1 \leq i \leq t)$ contains all vertices that are incident on the outer face obtained after the removal of the vertices in $V_1, \ldots, V_{i-1}$ in the *initial embedding* φ^+. Thus, *(peeling) step* i removes the vertices in V_i. Note that after each step i we obtain a unique embedding of the subgraph $G[V \setminus \bigcup_{j=1}^{i} V_j]$. Define the *peeling number* of a vertex $u \in V_i$ and an edge $\{u, v\}$ with $v \in V_j$ $(1 \leq i, j \leq t)$ in the peeling $\mathcal{P}$ as $\mathcal{N}_{\mathcal{P}}(v) = i + r(v)$, $\mathcal{N}_{\mathcal{P}}(\{u, v\}) = \min\{i, j\} + r(\{u, v\})$ and $\mathcal{N}_{\mathcal{P}}(G) = \max\{\max_{v \in V_G} \mathcal{N}_{\mathcal{P}}(v), \max_{e \in E_G} \mathcal{N}_{\mathcal{P}}(e)\}$. For simplicity, if r is clear from the context, let a peeling of (G, φ) be the peeling of $(G, \varphi, r, \emptyset)$.

The *weighted outerplanarity index* of G with weight function r is the minimal peeling number of the peelings of $(G, \varphi', r, \emptyset)$ over all rooted embeddings φ' of G. Note that the outerplanarity index of G is the weighted outerplanarity index in the special case $r \equiv 0$.

For a subgraph $H = (V_H, E_H)$ of G, let us say that two vertices u and v are connected in G by an *internally* H*-avoiding path* if there is a path from u to v in G, no edge of which belongs to H. Moreover, an H*-attached component* in G is a maximal vertex-induced subgraph $G' = G[V']$ of G such that each pair of vertices in V' is connected in G by an internally H-avoiding path. A subgraph H of G is a *peninsula* of G if $H = G$ or each H-attached component of G has at most two vertices in common with H. An *outer component* C of a peninsula H of G is defined in φ as H itself or as an H-attached component of G such that C contains an outside edge in φ. Next we define the induced graph $H(G)$ of a peninsula H of G and its embedding $\varphi_{H(G)}$ inherited from of the embedding φ of G. For an example, see Fig. 1. The motivation for this definition is that—as we observe later—peelings of (G, φ) and of $(H(G), \varphi_{H(G)})$ are very similar.

Definition 3 (Induced graph). *Given a connected planar graph G and a peninsula $H = (V_H, E_H)$ of G, the* induced graph $H(G)$ *has the vertex set V_H and the edge set $\{\{u, v\} \mid (\{u, v\} \in E_H) \vee (u, v \in V_H$ and u and v are connected in G by an internally H-avoiding path $)\}$. An edge $\{u, v\}$ in $H(G)$ is called a* virtual edge *if there is an internally H-avoiding path between u and v in G (even if $\{u, v\} \in E_H$).*

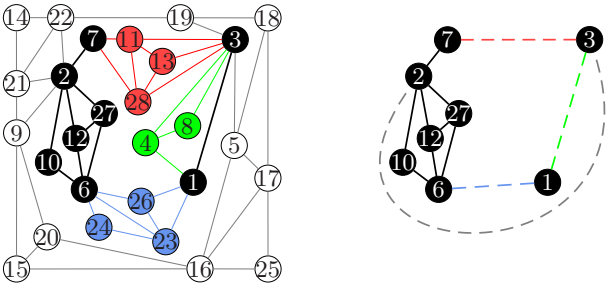


Fig. 1. Left: An embedded graph (G, φ) with a peninsula H (black), an outer component of H in G (all white vertices and vertices 2 and 3) and three further H-attached components in G. Right: The inherited embedding $\varphi_{H(G)}$ with the virtual edges dashed.

A *triconnected component* of G is the triconnected induced graph $H(G)$ of a peninsula H of G such that there exists no subgraph H' of G for which H is a subgraph of H' and $H'(G)$ is triconnected. For each peninsula H, define $\mathcal{R}_H$ as the function that maps each H-attached component C to the set of vertices common to C and H. Let C be a H-attached component. By definition of a peninsula, $|\mathcal{R}_H(C)| \leq 2$. If $|\mathcal{R}_H(C)| = 2$ we usually take $\mathcal{R}_H(C)$ as a virtual edge instead of a set of two vertices.

Definition 4 (Inherited embedding). *Given a connected embedded graph (G, φ) and a peninsula H of G, an* embedding φ' of H inherited from φ *is a rooted embedding of H for which the edges incident on each vertex v of H appear around v in the same order in φ and in φ'. Moreover, the outer face of φ' contains the outer face of φ.*

For the induced graph $H(G)$ of H, an inherited embedding *is a rooted embedding $H(G)$ defined almost as before; the only difference is that each virtual edge $\{u, v\}$ is mapped to the cyclic position around u of an edge incident on u of a simple internally H-avoiding path from u to v in G.*

Observe that the inherited embedding of an induced peninsula is unique if at each vertex u, a virtual edge $\{u, v\}$ can be mapped only to a set of consecutive edges around u, none of which is part of the peninsula. Let us call a peninsula H *good* if this is the case and if additionally for each virtual edge $\{u, v\}$ only one H-attached component C exists such that $\mathcal{R}_H(C) = \{u, v\}$. The last condition allows us later to identify each H-attached component with a virtual edge. For each embedded graph (G, φ) and for each a good peninsula H of G, define the *embedding* φ_H as the embedding of H inherited from φ. In the rest of this section we consider different properties of peelings and of inherited embeddings.

Lemma 5. *Let (G, φ) be a connected embedded graph and let e be an edge of G incident on the outer face in φ. For each good peninsula H of G, either e is in H and incident on the outer face in $\varphi_{H(G)}$, or e is in some H-attached component C in G and $\mathcal{R}_H(C)$ is incident on the outer face in $\varphi_{H(G)}$.*

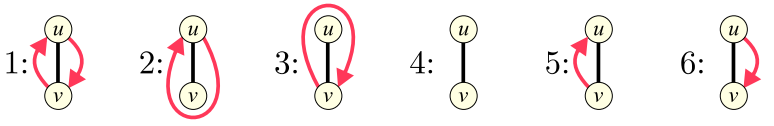


Fig. 2. Possibilities for an edge to be outside, e.g., 1: Both endpoints are outside

Proof. If e is in H, i.e., e is in $H(G)$, e remains outside. Otherwise, e is in $C \neq H$. Let $\{u, v\} = \mathcal{R}_H(C)$—possibly $u = v$. Let p a internally H-avoiding path from u to v using e such that p contains no simple cycle. The faces of φ are merged in $\varphi_{H(G)}$ such that at least one side of p is outside in $\varphi_{H(G)}$.

Given an embedded graph (G, φ), the *enclosure* of a vertex set S in G is the maximal subgraph C of G such that the only vertices outside in φ_C are the vertices of S.

Observation 6. *Let (G, φ) and (G', φ') be embedded graphs and let S and S' be vertex sets in these graphs, respectively, such that between the enclosure S^+_G of S in (G, φ) and the enclosure $S^+_{G'}$ of S' in (G', φ') an graph isomorphism f exists such that vertices u and v from S^+_G are adjacent if and only if $f(u)$ and $f(v)$ are adjacent in $S^+_{G'}$. Moreover, let r and r' be weight functions for G and G', respectively, such that $r(v) = r'(f(v))$ for all vertices v from S^+_G. If $\mathcal{N}_{\mathcal{P}}(v) - \mathcal{N}_{\mathcal{P}'}(f(v))$ is constant on all $v \in S$, then it is constant on all $v \in S^+_G$.*

Let (G, φ) be a connected embedded graph with weight function r and let $H = (V_H, E_H)$ be a good peninsula with an edge $\{u^*, v^*\}$ outside in φ. Informally, we now want to compare the peeling of (G, φ) with the peeling of the inherited embedding of $(H(G), \varphi_{H(G)})$. Moreover, what happens with an H-attached component $C = (V_C, E_C)$ of G during the peeling of φ?

Define $\mathcal{L}(V^+)$ for a set $V^+ = \{u\}$ as the list of edge sets $(\{(u, u)\}, \{\})$ and for a set $V^+ = \{u, v\}$ as $(\{(u, v), (v, u)\}, \{(u, u)\}, \{(v, v)\}, \{\}, \{(v, u)\}, \{(u, v)\})$. The directed edges of Fig. 2 shows $\mathcal{L}(\{u, v\})$. Take E^+ as a set in the list $\mathcal{L}(\{u^*, v^*\})$ and $\mathcal{P} = (V_1, V_2, \ldots, V_t)$ as a peeling of (G, φ, r, E^+). Choose u and v such that $\{u, v\} = \mathcal{R}_H(C)$—possibly $u = v$. If $u \in V_i$ and $v \in V_j$ $(1 \leq i, j \leq t)$, take $q = \min\{i, j\}$. For the embedding φ_C of C inherited from φ, define $\mathcal{S} = Out(\varphi_C)$. Let H' be the outer component of C that is a supergraph of H and that contains $\{u^*, v^*\}$. Let $\mathcal{P}'$ be the peeling of $(H'(G), \varphi_{H'(G)}, r', E^+)$ where r' is an arbitrary weight function such that r' is equal to r for all vertices of $H'(G)$. The following two properties are proved below.

Property 1. $V_q \cap V_C$ equals one of the 6 sets below. In other words, if we ignore the vertices not in C, peeling step q removes one of these 6 sets: 1. $\{u, v\}$, 2. $\{u\}$, 3. $\{v\}$, 4. $\mathcal{S}$, 5. $\mathcal{S}^{\varphi_C}_{u \to v}$ or 6. $\mathcal{S}^{\varphi_C}_{v \to u}$. The remaining vertices of $\mathcal{S}$ are removed in step $q + 1$.

Property 2. The removal of C from G if $|\mathcal{R}_H(C)| = 1$ and the replacement of C by the (virtual) edge $\mathcal{R}_{H'}(C)$ in G if $|\mathcal{R}_H(C)| = 2$ does not change the peeling numbers of the vertices in H', i.e., $\mathcal{N}_{\mathcal{P}}(H') = \mathcal{N}_{\mathcal{P}'}(H')$.

For each of the 6 sets of Property 1 an example in Fig. 2 shows a situation just before the *qth* peeling step. In each example the black edge corresponds to C and each directed edge has a corresponding undirected path in H.

Proof of Property 1. Consider the two cases $|\mathcal{R}_H(C)| = 1$ and $|\mathcal{R}_H(C)| = 2$. For the first case (i.e., $u = v$), observe that if one vertex of $\mathcal{S} \setminus \{u\}$ is removed in peeling step q, then all vertices in $\mathcal{S}$ are removed in that step because there is no edge from $\mathcal{S} \setminus \{u\}$ to H. Thus, $V_q \cap V_C$ equals $\{u\}$ or $\mathcal{S}$. In the second case we argue in the same way that if a vertex in $\mathcal{S}^{\varphi_C}_{u \to v} \setminus \{u, v\}$ or $\mathcal{S}^{\varphi_C}_{v \to u} \setminus \{u, v\}$ is removed, then all vertices in $\mathcal{S}^{\varphi_C}_{u \to v}$ or $\mathcal{S}^{\varphi_C}_{v \to u}$, respectively, are removed. The remaining vertices in $\mathcal{S}$ are outside after step q.

Proof of Property 2. Starting with the initial embedding $\varphi_0^{\mathcal{P}} = \varphi$ and $\varphi_0^{\mathcal{P}'} = \varphi_{H'(G)}$ and carrying out in parallel the peelings $\mathcal{P}$ and $\mathcal{P}'$, let us compare the embeddings $\varphi_i^{\mathcal{P}}$ and $\varphi_i^{\mathcal{P}'}$ obtained after $i = 1, 2, \ldots$ peeling steps. By induction on i, we can observe that if a *C-internal face* is one that is incident only on vertices of C, there is the following bijection between the not C-internal faces of $\varphi_i^{\mathcal{P}}$ and the faces of $\varphi_i^{\mathcal{P}'}$: Each face f in $\varphi_i^{\mathcal{P}}$ is mapped to the face in $\varphi_i^{\mathcal{P}'}$ whose boundary vertices are those of f, except for the vertices in C. Thus, we can conclude that a face of $\varphi_i^{\mathcal{P}}$ is the outer face if and only if the corresponding face of $\varphi_i^{\mathcal{P}'}$ is the outer face, i.e., a vertex in H' is outside after the same number of peeling steps in $\mathcal{P}$ and in $\mathcal{P}'$.

Since C is the enclosure of $\mathcal{S}$, we can apply Observation 6. Informally, the peeling numbers of vertices in C depend only on the peeling numbers of vertices in $\mathcal{S}$. Because of that and Property 1, a set of edges $E' \in \mathcal{L}(\{u, v\})$ exists such that for the peeling $\mathcal{P}''$ of (C, φ_C, r, E') and all $v \in C$: $\mathcal{N}_{\mathcal{P}}(v) = \mathcal{N}_{\mathcal{P}''}(v) + q - 1$. Observe that E' can be determined during the peeling $\mathcal{P}'$ of $(H'(G), \varphi_{H'(G)}, r', E^+)$ by observing which parts of the virtual edge $\mathcal{R}_H(C)$ are outside just before it is removed. For that reason, let us define E' as the *induced set of extra edges* of $\mathcal{P}'$ and $\mathcal{R}_H(C)$. Figure 2 shows various possibilities for how a virtual edge (black) can be outside.

The removal and replacement described in Property 2, applied to all H-attached components of G in φ, results (after the removal of multiple edges) in the inherited embedding $\varphi_{H(G)}$. Thus, for the vertices in H, the peeling numbers in the peeling of (G, φ, r, E^+) and in the peeling $\mathcal{P}^*$ of $(H(G), \varphi_{H(G)}, r', E^+)$ coincide for each set E^+ where r' is—except to the following two changes—equal to r. The changes enables us to add the peeling numbers of all H-attached components to the computation of the peeling $\mathcal{P}^*$. The first change is to extend the domain of the weight function r to all virtual edges $\mathcal{R}_H(C')$ and set $r(\mathcal{R}_H(C'))$ to -1 plus the peeling number of C' in the peeling $(C', \varphi_{C'}, r, E')$ where E' is the *induced set of extra edges* of $\mathcal{P}^*$ and $\mathcal{R}_H(C')$ and the second change is to increase $r(v)$ by -1 plus the highest peeling number of an H-attached component C^* with $\mathcal{R}_H(C^*) = \{v\}$ $(v \in V_H)$. Leave $r(v)$ unchanged if no such C^* exists. Denote the function obtained the $(G, H, \mathcal{P}^*)$*-extended and* (G, H)*-increased weight function* of r.

Given a graph G, a set of directed edges E' and a set S of vertices (edges), an embedding φ is *vertex-constrained* (*edge-constrained*) *optimal* for (G, r, E', S) if

all $s \in S$ are outside with respect to φ and for all embeddings φ' the following is true: Either not all $s \in S$ are outside with respect to φ' or $\mathcal{N}_{\mathcal{P}'}(G) \leq \mathcal{N}_{\mathcal{P}''}(G)$, where $\mathcal{P}'$ and $\mathcal{P}''$ are the peelings of (G, φ, r, E') and (G, φ', r, E'), respectively. Note that a smallest peeling number and a rooted embedding with this peeling number can be computed simultaneously.

Theorem 7. *Given a graph $G = (V_G, E_G)$ with a weight function r and an edge $\{u^*, v^*\} \in E_G$, one can find an edge-constrained optimal embedding of $(G, r, E^+, \{u^*, v^*\})$ for all $E^+ \in \mathcal{L}(\{u^*, v^*\})$ as follows:*

Taking a good peninsula $H = (V_H, E_H)$ of G that contains $\{u^*, v^*\}$, first determine for each H-attached component C the constrained optimal embedding of $(C, r, E', \mathcal{R}_H(C))$ for all $E' \in \mathcal{L}(\mathcal{R}_H(C))$ recursively.

For each E^+ and each embedding φ of $H(G)$ with $\{u^*, v^*\}$ outside, then determine the peeling number of the peeling $\mathcal{P}$ of $(H(G), \varphi, r', E^+)$, where r' is the $(G, H, \mathcal{P})$-extended and (G, H)-increased weight function of r.

For each E^+ finally return the smallest peeling number and a corresponding embedding φ.

4 The Outerplanarity Index for Biconnected Graphs

Before applying Theorem 7, we have to decompose G into good peninsulas. Given a biconnected planar graph G, this can be done in linear time by constructing a so-called SPQR tree of G [11]. An *SPQR tree* [6] of a biconnected graph G is a tree $\mathcal{T} = (\mathcal{W}, \mathcal{F})$ with a mapping $\mathcal{M}$ from $\mathcal{W}$ to a set of so-called *split-components* of G. Each split-component—consisting of either a *triconnected*, a *cycle* or a *multiple-edge component* as defined later—is the induced graph $H(G)$ of a good peninsula H of G, which, in the case of a multiple-edge component, is extended by some edges. Additionally, for an SPQR tree the following two properties hold: First, each edge of G is part of exactly one split-component $H \in \mathcal{M}(\mathcal{W})$ and second, two nodes $w_1, w_2 \in \mathcal{W}$ with $H_i = \mathcal{M}(w_i)$ $(i = 1, 2)$ are connected by an edge in $\mathcal{F}$ if and only if $\mathcal{R}_{H_1}(H_2) \neq \emptyset$. A cycle component is a simple cycle and a multiple-edge component consists of only 2 vertices (a separation pair of G) connected by at most one edge of G and several virtual edges. In addition, let us extend the mapping $\mathcal{M}$ from the nodes of $\mathcal{T}$ to each subtree T' of $\mathcal{T}$ in the canonical way. In detail, if we take $\{w_1, \ldots, w_\ell\}$ as the nodes of T', $(W_i, F_i) = \mathcal{M}(w_i)$ and S as the set of all virtual edges $\{e \mid e \in F_i \cap F_j$ and $(1 \leq i < j \leq \ell)\}$, define $\mathcal{M}(T') = (\cup_{i=1}^{\ell} W_i, \cup_{i=1}^{\ell} F_i \setminus S)$.

For a graph G and an edge e of G, a *rooted SPQR tree* $\mathcal{T} = (\mathcal{W}, \mathcal{F})$ of (G, e) is an SPQR tree of G rooted at the unique node $r \in W$ with $e \in \mathcal{M}(r)$. The *subtree module* of a node w in $\mathcal{T}$ is the graph $\mathcal{M}(T_w)$, where T_w is the maximal subtree of $\mathcal{T}$ with root w. Observe that in a rooted SPQR tree each subtree module is the induced graph of a good peninsula. Let $w \in \mathcal{W}$ and let $C = \mathcal{M}(w)$. The *virtual parent-edge* e_p of C and of the subtree module of w is defined as follows. If w is the root of a rooted SPQR tree of (G, e), take $e_p = e$; otherwise, take $e_p = \mathcal{R}_C(H)$ with $H = \mathcal{M}(w')$ and w' being the parent of w.

Given a biconnected planar graph G, a weight function r and an edge e_{out} of G, let us now compute an edge-constrained optimal embedding of G with e_{out} outside since we can later use this computation as an auxiliary procedure for finding an optimal or an approximative embedding. Recall that a rooted embedding φ of G is optimal (c-approximative) if the peeling steps of the peeling of $(G, \varphi, r, \emptyset)$ equals (is bounded by c times) the outerplanarity index of G. Note that the peeling steps do not depend on r.

First, we compute a rooted SPQR tree $\mathcal{T}$ of (G, e_{out}) and then traverse it bottom-up. Let w be the current node visited by the traversal and let $e_p = \{u, v\}$ be the virtual parent edge of w. Define $H = \mathcal{M}(w)$, C as the subtree module of w and φ as some constrained optimal embedding of $(G, r, \emptyset, e_{\text{out}})$. In the following we determine an embedding $\varphi_{C'}$ of $C' = C - e_p$ inherited from φ. Let $H' = H - e_p$. By Lemma 5, e_p is incident on the outer face of φ_C, i.e., u and v are incident on the outer face of $\varphi_{C'}$. By Theorem 7, for all $E^+ \in \mathcal{L}(\{u, v\})$ we can determine a constrained optimal embedding $\varphi_{C'}$ of (C', r, E^+, e_p) by considering the peeling $\mathcal{P}$ of $(H', \varphi_{H'}, r', E^+)$ for all embeddings $\varphi_{H'}$ of H' with e_p outside and r' the $(C', H', \mathcal{P})$-extended weight function of r. The reason for this is that recursive calls have already found the constrained optimal embeddings for the subtree modules of all children of w.

If H is a triconnected or a cycle component, there are only two possible rooted embeddings of H with e_p outside. Thus, there is only one rooted embedding of H' with u and v outside. Therefore, $\varphi_{H'}$ and also $\varphi_{C'}$ can be found in time linear in the size of H. If $H = (\{u, v\}, E_H)$ is a multiple-edge component, both vertices of H have to be outside. Depending on E^+, we can choose $d \in \{0, 1, 2\}$ children of w such that their subtree modules are (with an edge) outside in the embedding obtained from $\varphi_{C'}$ by adding E^+ clockwise. Observe that the peeling numbers of a subtree module of a child of w differ only by at most one for different orders of the embeddings of the subtree modules between u and v. Let $C_1, \ldots, C_\ell$ be the subtree modules of the children of w. The *inside peeling number* of C_i ($i \in \{1, \ldots, \ell\}$) is the number of peeling steps in a constrained optimal embedding of $(C_i, r, \{(u, v), (v, u)\}, \{u, v\})$ and the *outside peeling number* of C_i is the smaller number of peeling steps in the constrained optimal embeddings of $(C_i, r, \{(u, v)\}, \{u, v\})$ and of $(C_i, r, \{(v, u)\}, \{u, v\})$. Since the peeling number of C' is the maximum over the peeling numbers of all C_i ($1 \leq i \leq \ell$), our goal is to reduce the largest peeling number. Therefore, we choose for C' an embedding such that d subtree modules of the children of w with largest inside peeling number have outside vertices in $\varphi_{C'}$. Finding d subtree modules with largest inside peeling number (i.e., finding $\varphi_{C'}$) needs time linear in the size of H.

Lemma 8. *Given a biconnected graph G with a weight function r and an edge e_{out} of G, an edge-constrained optimal embedding of $(G, r, \emptyset, e_{\text{out}})$ can be found in linear time.*

For an optimal embedding of a given graph $G = (V, E)$, iterate over all edges $e \in E$ and for an approximative embedding of G consider only one arbitrary edge $e \in E$. In each iteration use weight function $r \equiv 0$ and compute an edge-constrained optimal embedding with e outside. The quality of the approximative

embedding φ with edge e outside can be estimated with arguments similar to those used in the proof of Corollary 2. Let φ^{OPT} be an optimal embedding of G and let f^{OPT} be the outer face of φ^{OPT}. Choose φ' as the rooted embedding such that φ^{OPT} and φ' have the same combinatorial embedding and e is outside. The number of peeling steps to remove (G, φ') is at most twice the number of peeling steps to remove $(G, \varphi^{\mathrm{OPT}})$. Since φ is an edge-constrained optimal embedding with $r \equiv 0$ and e outside, (G, φ) does not need more peeling steps to remove than (G, φ').

Corollary 9. *Given a biconnected planar graph G, a 2-approximative and an optimal embedding of G can be found in linear and in quadratic time, respectively.*

5 The Outerplanarity Index for General Graphs

Given a planar graph $G = (V_G, E_G)$ with weight function r, we can determine the peeling number for each connected component separately. Thus, let us assume in the following that all graphs considered are connected.

Let $e \in E_G$ and let B be the biconnected component containing e. For the moment, let us assume that we know the peeling number of a vertex-constrained optimal embedding $(H, r, \emptyset, \mathcal{R}_H(B))$ of each B-attached component H. By Theorem 7 we can then determine the (G, B)-increased weight function r' of r and use the algorithm for edge-constrained optimal embeddings on B with weight function r' to obtain an edge-constrained optimal embedding for the whole graph G with e outside. By an iteration over all edges of G an optimal embedding can be found.

It remains to show how all necessary vertex-constrained optimal embeddings can be found. For this purpose let us construct the block-cutpoint tree of G [7,10]. A *block-cutpoint tree* of a connected graph G is a tree T whose nodes are the separation vertices and the biconnected components of G. Each biconnected component B is incident in T to all separation vertices in B. A block-cutpoint tree can be found in linear time [18]. A *rooted block-cutpoint tree* $T = (W, F, R)$ is a block-cutpoint tree (W, F) rooted at some biconnected component R. Given a rooted block-cutpoint tree $T = (W, F, R)$, the *subtree* (*supertree*) component of a biconnected component B of G is the subgraph of G consisting of the biconnected component B and all biconnected components below (strictly above) B in T.

Let us traverse some rooted block-cutpoint tree $T = (W, F, R)$ of G first bottom-up and then top-down. During the bottom-up (top-down) traversal we compute at each biconnected component $B \neq R$ with parent v a vertex-constrained optimal embedding of the subtree (supertree) component of B with v outside as follows. Define during the traversals for each B in the bottom-up case B' as B and in the top-down case B' as the grandparent of B. Note that for each sub- and supertree component C, we already know the peeling numbers of each B'-attached component in C. Thus, we can determine the (C, B')-increased weight function r' of r. A vertex-constrained optimal embedding with a vertex v outside can be found by iterating over each edge e of B' incident to v and computing an edge-constrained optimal embedding $(B', r', \emptyset, e)$. In the end, we obtain for each separation vertex v a vertex-constrained optimal embeddings

of all $(\{v\},\{\})$-attached components in G in quadratic time. Taking $r \equiv 0$ we obtain the following.

Theorem 10. *Given a planar graph G, an optimal embedding and the outerplanarity index of G can be found in quadratic time.*

For an approximate embedding of a graph G with weight function r the idea is so far to fix an arbitrary edge e and to search for an edge-constrained optimal embedding with e outside. Unfortunately, we cannot use the same approach as for an optimal embedding since determining a vertex-constrained optimal embedding of a single big biconnected component may take to much time. Therefore we take a rooted block-cutpoint tree T, compute only bottom-up for each biconnected component B with parent v and subtree component $C = (V_C, E_C)$ in T the two embeddings φ and φ' of B as defined below and return the smaller peeling number of the two peelings of (B,φ) and of (B,φ) and the corresponding embedding. As weight function use the (C,B)-increased weight function r' of r adapted from the peeling numbers obtained by recursive calls.

For some edge e incident on v take φ as an edge-constrained optimal embedding of C with e outside. If D is a grandchild of B such that the algorithm has recursively obtained on input D a biggest peeling number over all grandchildren and if v^* is the parent of D, φ' is taken to be a vertex-constrained optimal embedding of C with v and v^* outside. Note that φ' need not exist. We can find φ'—if it exists—by computing an edge-constrained optimal embedding of $(V_C, E_C \cup \{v,v^*\})$ with $\{v,v^*\}$ outside and then removing $\{v,v^*\}$. Use the results of the recursive calls to find φ and φ' in time linear in the size of B.

We now show by induction on the height of a vertex B in T that this bottom-up computation finds a *vertex-constrained 2-approximative embedding* φ of C with v outside, i.e., the peeling number of (C,φ) is at most twice the peeling number of (C,φ^*) for every other embedding φ^* of C with v outside.

Let k_B^{OPT}, k_C^{OPT} and k_D^{OPT} be the peeling number of a constrained optimal embedding of $(B,r,\emptyset,\{v\})$, $(C,r,\emptyset,\{v\})$ and $(D,r,\emptyset,\{v\})$, respectively, and let k_B, k_C and k_D be the peeling numbers obtained from our algorithm for inputs B, C and D, respectively. Although in φ we enforce one fixed edge incident to v to be outside, the extra peeling costs of $k_B - k_B^{\mathrm{OPT}}$ are at most 1 since after a first peeling step the outer face contains each face incident on v. If a vertex of B causes peeling number k_C, i.e., $k_C = k_B$, the embedding obtained for C is vertex-constrained 2-approximative since $k_C = k_B \leq k_B^{\mathrm{OPT}} + 1 \leq 2k_B^{\mathrm{OPT}} \leq 2k_C^{\mathrm{OPT}}$. Otherwise, some subtree component H—possibly $H = D$—of some grandchild of B causes the peeling number k_C. Let v' be a child of B connecting B and H. Let k_H^{OPT} be the peeling number of a constrained optimal embedding of $(H,r,\emptyset,\{v\})$ and let k_H be the peeling number obtained from our algorithm for input H. Then $k_D \geq k_H$. Define $q = k_C^{\mathrm{OPT}} - k_H^{\mathrm{OPT}}$. Let us consider 3 cases:

$\mathbf{q > 0}$: $k_C \leq 2k_H^{\mathrm{OPT}} + (q+1) \leq 2k_H^{\mathrm{OPT}} + 2q = 2k_C^{\mathrm{OPT}}$.
$\mathbf{q = 0 \wedge k_H < 2k_H^{OPT}}$: $k_C \leq k_H + (q+1) = (k_H+1) + q \leq 2k_H^{\mathrm{OPT}} + 2q = 2k_C^{\mathrm{OPT}}$.
$\mathbf{q = 0 \wedge k_H \geq 2k_H^{OPT}}$: $k_D^{\mathrm{OPT}} \geq k_D/2 \geq k_H/2 \geq k_H^{\mathrm{OPT}} = k_C^{\mathrm{OPT}}$, so that in some optimal embedding v^* is outside.

In the first and second case φ and in the last case φ' gives us a vertex-constrained 2-approximative embedding. Similarly as in the proof of Corollary 9 we can conclude that a vertex-constrained 2-approximative embedding is a 4-approximative embedding. Taking $r \equiv 0$ we obtain the following.

Theorem 11. *Given a planar graph G, a 4-approximative embedding of G and a 4-approximation of the outerplanarity index of G can be found in linear time.*

References

1. Baker, B.S.: Approximation algorithms for NP-complete problems on planar graphs. Journal of the ACM 41, 153–180 (1994)
2. Bienstock, D., Monma, C.L.: On the complexity of embedding planar graphs to minimize certain distance measures. Algorithmica 5, 93–109 (1990)
3. Chiba, N., Nishizeki, T., Abe, S., Ozawa, T.: A linear algorithm for embedding planar graphs using PQ-trees. J. Comput. System Sci. 30, 54–76 (1985)
4. Bodlaender, H.L.: A linear time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. 25, 1305–1317 (1996)
5. Bodlaender, H.L.: A partial k-arboretum of graphs with bounded treewidth. Theoretical Computer Science 209, 1–45 (1998)
6. Di Battista, G., Tamassia, R.: Incremental planarity testing. In: Proc. 30th IEEE Symp. on Foundations of Computer Science, pp. 436–441 (1989)
7. Gallai, T.: Elementare Relationen bezüglich der Glieder und trennenden Punkte von Graphen. Magyar Tud. Akad. Mat. Kutato Int. Kozl. 9, 235–236 (1964)
8. Garey, M.R., Johnson, D.S: Computers and Intractability. A Guide to the Theory of NP-Completeness. W. H. Freeman and Co, San Francisco, Calif. (1979)
9. Gu, Q.P., Tamaki, H.: Optimal branch-decomposition of planar graphs in $O(n^3)$ time. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 373–384. Springer, Heidelberg (2005)
10. Harary, F., Prins, G.: The block-cutpoint-tree of a graph. Publicationes Mathematicae Debrecen 13, 103–107 (1966)
11. Gutwenger, C., Mutzel, P.: A linear time implementation of SPQR tree. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 77–90. Springer, Heidelberg (2001)
12. Mehlhorn, K., Mutzel, P.: On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. Algorithmica 16, 233–242 (1996)
13. Reed, B.: Finding approximate separators and computing tree-width quickly. In: Proc. 24th Annual ACM Symp. on Theory of Computing (STOC 1992), pp. 221–228 (1992)
14. Röhrig, H.: Tree Decomposition: A Feasibility Study, Master's thesis, Max-Planck-Institut für Informatik in Saarbrücken (1998)
15. Robertson, N., Seymour, P.D.: Graph minors. I Excluding a forest. J.Comb.Theory Series B 35, 39–61 (1983)
16. Robertson, N., Seymour, P.D.: Graph minors. III Planar tree-width. J.Comb.Theory Series B 36, 49–64 (1984)
17. Seymour, P.D., Thomas, R.: Call routing and the ratcatcher. Combinatorica 14(2), 217–241 (1994)
18. Tarjan, R.E.: Depth-first search and linear algorithms. SIAM J. Comp. 1, 146–160 (1972)
19. Whitney, H.: Non-separable and planar graphs. Trans. Amer. Math. Soc. 34, 339–362 (1932)

On the Size of Succinct Indices*

Alexander Golynski[1], Roberto Grossi[2], Ankur Gupta[3],
Rajeev Raman[4], and Satti Srinivasa Rao[5]

[1] David R. Cheriton School of Computer Science, University of Waterloo, Canada
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Butler University, USA
[4] Department of Computer Science, University of Leicester, UK
[5] IT University of Copenhagen, Denmark

Abstract. A succinct data structure occupies an amount of space that is close to the information-theoretic minimum plus an additional term. The latter is not necessarily a lower-order term and, in several cases, completely dominates the space occupancy both in theory and in practice. In this paper, we present several solutions to partially overcome this problem, introducing new techniques of independent interest that allow us to improve over previously known upper and lower bounds.

1 Introduction

What is the minimal space complexity for encoding and storing a subset of n elements out of an universe of m elements? Having numbered the elements of the universe from 1 to m, information theory indicates that since there are $\binom{m}{n}$ such subsets, each one to be distinguished from the others requires at least $B(m,n) = \lceil \lg \binom{m}{n} \rceil$ bits[1]. According to Jacobson [9], a representation that uses $B(m,n)+O(1)$ bits for an arbitrarily chosen subset is called a *succinct encoding*.

Dictionaries over the universe $\{1, 2, \ldots, m\}$ are ubiquitous data structures that have been the subject of deep theoretical investigation and are interesting also from a practical point of view. A subset implicitly represents a dictionary storing n elements as a bit-string S of length m with n **1**s, such that the ith bit is **1** if and only if the ith element of the universe belongs to the dictionary. We wish to support the following operations, where x can be chosen as bit **0** or **1**:

- $\mathsf{rank}_x(S,i)$ returns how many xs are contained in the first i positions of S.
- $\mathsf{select}_x(S,i)$ returns the position of the ith occurrence of x in S.

We omit the first parameter S when it is clear from the context. We wish to support these operations in *constant* time on the RAM model with word size $O(\lg m)$ bits. Following [15], we call such a data structure a *fully indexable dictionary* (FID). Note that the predecessor of an element k to the dictionary can be found in $O(1)$ time using $\mathsf{select}_1(\mathsf{rank}_1(k))$. Using the lower bound by Beame and Fich [1], we can infer that FIDs using at most $B(m,n) + n^{O(1)} \lg m$ bits do not

* Part of this work was done while Gupta was at Purdue University, and Raman was on study leave and visiting DIMACS.

[1] lg denotes the logarithm base 2.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 371–382, 2007.

exist in general. A recent paper by Patrascu and Thorup [14] implies that there is no FID that uses $B(m,n)+o(n)$ bits unless $m = n\lg^{O(1)} n$ or $n = \lg^{O(1)} m$.

Theorem 1 ([15]). *A FID uses* $B(m,n) + O(m \lg\lg m/\lg m)$ *bits.*

Theorem 1 has found many applications, but leaves open the fundamental question of the precise space complexity of FIDs that support operations quickly — as a first step, is the term of $O(m \lg\lg m/\lg m)$ in Theorem 1 necessary?

In an effort to answer this question, researchers have considered the case when the bit-string of m bits representing the set is explicitly stored in the encoding in raw form: such encodings are called *systematic* in [4]. Note that in many applications of FIDs, $n \sim m/2$, and $B(m,n) = m - O(\lg m)$. In this case, the restriction to systematic encodings would appear, naively, not to be a serious one. For systematic encodings, it has been shown that $\Omega(m \lg\lg m/\lg m)$ additional bits are needed to support operations in $O(1)$ time by Miltersen [10] (for rank) and Golynski [6] (for select). A matching upper bound, with improved constants is described in [6]. Hence, systematic FIDs require $m + \Theta(m \lg\lg m/\lg m)$ bits.

As $B(m,n)$ can be significantly less than m when $n \ll m$, it is clear that any FID that approaches the information-theoretic lower bound of $B(m,n)$, including the FID of Theorem 1, cannot be systematic. We show the following:

(1) We give an FID that uses $B(m,n) + O(m \lg\lg m/(\lg m)^2)$ bits of space, saving a $\lg m$ factor from the "wasted space" in Theorem 1. FIDs are employed in a variety of compressed data structures for text indexing, data compression, and graph algorithms: we improve the complexity for many of these results. Since $B(m,n) \leq m$ for all n, we obtain an FID that takes no more than $m + O(m \lg\lg m/(\lg m)^2)$ bits of space, which is lower than the lower bound of Miltersen [10] and Golynski [6] for systematic encodings. This shows that even in the important case of FIDs for sets where $n \sim m/2$ (and hence where $B(m,n) \sim m$), the restriction to systematic encodings yields weaker space upper bounds. The general observation that systematic encodings are strictly weaker than unrestricted encodings was previously noted in several places, most notably Gál and Miltersen [4] and Munro et al. [12] (see also [15]).

(2) As noted, $O(1)$-time FIDs only exist when $m = n\lg^{O(1)} n$. For sufficiently sparse sets, e.g., $m = n(\lg n)^{1+c}$ for any $c > 0$, the $O(m \lg\lg m/\lg m)$ term in Theorem 1 is larger than $B(m,n)$, and Theorem 1 is not even close to being a succinct representation. A previous result of Pagh [13] obtained an implementation of rank alone using $B(m,n) + O(n(\lg\lg n)^2/\lg n)$ bits for the case where $m = n\lg^{O(1)} n$. In this paper, we provide an implementation of an FID (including select) using $B(m,n) + O(n(\lg\lg n)^2/\lg n)$ bits when $m = n\lg^{O(1)} n$, thus providing a succinct FID for all "interesting" values of n for which $O(1)$-time FIDs can exist. The lower bound of $\Omega((n \lg t)/t)$ in [6] implies that for $m = n\lg^{O(1)} n$ and $t = O(\lg m) = O(\lg n)$, we obtain $\Omega(n \lg\lg n/\lg n)$, so our upper bound is a $\lg\lg n$ factor larger (and can be converted to an indexing data structure).

(3) *Informative encoding* is a crucial and novel technique that we introduce in this paper. It is able to store the summary information by "wasting" only $O(1)$ bits. As an example, we give an encoding E of a sequence of k bits that

occupies at most $k+2$ bits, such that the number of **1**s in the sequence can be determined by looking at $\lg k + O(1)$ consecutive bits of E. Instead, existing FIDs use $k + \Theta(\lg k)$ bits for this purpose, giving rise to $O(m \lg\lg m / \lg m)$.

(4) We also study the complexity of encoding a sequence of m balanced parentheses, to support the following operations in $O(1)$ time: finding a matching parenthesis (findclose, findopen) and finding the nearest enclosing matching pair of parentheses (enclose). This data structure is fundamental to succinct representations of trees and graphs [11,9]. We obtain an upper bound of $m + O(m \lg\lg m/(\lg m)^2)$ bits — the lower-order term is $\Theta(\lg m)$ times smaller than [5] — and a lower bound of $m + \Omega(m \lg\lg m / \lg m)$ for systematic encodings.

We aim at improving the additional terms. Some experimental work has shown that this improvement is significant [8]. To this end, let's discuss one of the applications of succinct dictionaries, namely, compressed text indexing [7]. Given a text T defined over an alphabet of size σ, its compressed text index or self-index stores T in $nH_k + O(n \lg\lg n / \lg_\sigma n)$ bits for $k \leq \epsilon \lg_\sigma n$ and a positive constant $\epsilon < 1$, and allows searching for any pattern string P as a substring of T in $O(|P| + \lg^{O(1)} n)$ time. The additional term of $O(n \lg\lg n / \lg_\sigma n)$ can be much larger than nH_k (i.e., if T is made up of all equal symbols, $nH_k = 0$), and comes from the additional term of $O(m \lg\lg m / \lg m)$ in the FIDs used to implement the compressed text index. Improving the latter to $O(m \lg\lg m / \lg^2 m)$ may give a better bound of $nH_k + O(n \lg\lg n / (\lg_\sigma n \lg n))$ for the space complexity.

Preliminaries. We describe a few basic notations and conventions that we will use throughout the paper. For any integer $x \geq 1$, we let $[x]$ refer to the set $\{1, 2, \ldots, x\}$. For a given input bit-string S, we will often partition S into fixed-length substrings, which we call *blocks*; S may also be divided into variable-length *chunks*. We refer to the length of a chunk (in bits) as its *width*, and the number of **1**s it contains as its *cardinality*.

2 Compressed Bit Vector

In this section, we improve the lower order term in Theorem 1 by a poly-logarithmic factor when the length m of a bit vector S is a poly-logarithmic factor larger than its cardinality n, i.e., $m = n \lg^{O(1)} n$.

Theorem 2. *When $m = \Theta(n \lg^c n)$ for a constant $c > 0$, we can store a FID using $B(m,n) + O(n (\lg\lg n)^2 / \lg n)$ bits.*

2.1 Partition into Chunks

We partition S into blocks of length $b = m(\lg n)/n$, and further partition blocks into variable width chunks. We allocate chunks within a block in a greedy fashion from left to right subject to the constraint that the cardinality of a chunk cannot exceed $s = \lg n / (2(c+1) \lg\lg n)$. See Fig. 1 for an example.

Lemma 1. *The number g of chunks $c_1, \ldots, c_g$ obtained by the above greedy partitioning satisfies $g \leq n/s + m/b = O(n \lg\lg n / \lg n)$.*

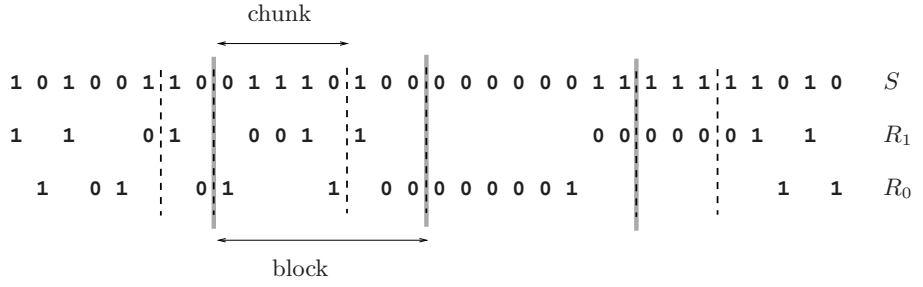


Fig. 1. Partition into chunks (dashed lines) of sequence S and its corresponding bitvectors R_0 and R_1, using the logical blocks (in gray lines)

Let m_j denote the width of chunk c_j, and n_j denote the cardinality of c_j, for $1 \leq j \leq g$. Note that $m = \sum_{j=1}^{g} m_j$ and $n = \sum_{j=1}^{g} n_j$, and that $m_j \leq b$ and $n_j \leq s$. We build a basic suite of common components according to Lemma 1.

The *chunk representation* bit-string C: each chunk c_j is represented using the canonical subset encoding in $B(m_j, n_j) \leq s \lg b + O(1) = (\lg n)/2 + O(1)$ bits of space, and C is the orderly concatenation of the binary encoding for $c_1, c_2, \ldots, c_g$. Using the fact that $\sum_{j=1}^{g} B(m_j, n_j) \leq B(m, n) + O(g)$ (see [2]), the total space occupancy for C is $B(m, n) + O(g)$ bits.

The *chunk offset* data structure CO: it stores, for each chunk c_j, the starting position of its binary encoding in C. The data structure is an array with fixed-size entries, where the jth entry stores the length of the binary encoding of c_j, for $1 \leq j \leq g$. The size of c_j's binary encoding cannot be greater than m_j; since $m_j \leq b$, the array entries can take $\lg b$ bits each. In addition, we explicitly store the offset of every $(\lg n/(2 \lg b))$th chunk in a second array, and build a lookup table for every possible sequence of $(\lg n/(2 \lg b))$ chunk offsets. This allows us to compute the prefix sums of the offsets for the first k chunks in the sequence, where $1 \leq k \leq (\lg n/(2 \lg b))$. The total space used by the two arrays and the table is $O(g \lg b) = O(n(\lg \lg n)^2 / \lg n)$ bits. To decode an individual offset, we look up an explicitly-stored offset in the second array, and then perform a table lookup on the entries of the first array in between.

2.2 Supporting rank on S

We store the following data structures. We store *chunk widths* $m_1, m_2, \ldots, m_g$ in an array CW with fixed-size entries of $\lg b$ bits each, taking $O(n(\lg \lg n)^2 / \lg n)$ bits. First, we find the correct *block* of S by using a classic two-level scheme that occupies $O(n \lg \lg n / \lg n)$ bits of space. To compute prefix sums within a block, we store a sequence of implicit tree structures, one for each block, that performs this search in $O(1)$ time. To mark the location of each implicit tree, we require $O(g \lg \lg n)$ bits, using Theorem 1.

Assume that b is a multiple of $\lceil \lg b \rceil$. Then, the implicit tree for a given block has no more than $b / \lg b = O(\lg^{c+1} n / \lg \lg n)$ leaves. The internal nodes are fixed-size of $(\lg n)/2$ bits each, and have $\Theta(\lg n / \lg b) = \Theta(\lg n / \lg \lg n)$ children. An internal node stores the sum of the chunk widths stored in the descending leaves

of each child using $\lg b$ bits per child. These stored sums fit in the node. The total number of bits needed to store the internal nodes is proportional to the number of bits required to store the chunk widths, which are the leaves. Therefore, the space occupancy for *all* implicit trees is $O(g \lg\lg n)$, as for the sequence of chunk widths. All the nodes are stored in heap order. Thus, multiway branching from each node can be performed in $O(1)$ time by a table lookup using an additional $O(\sqrt{n} \cdot b) = \Theta(\sqrt{n} \lg^{c+1} n)$ bits. Finally, since the height of each tree is $O(1)$, we can identify the correct leaf/chunk c_j in $O(1)$ time. The total space required by CW is $O(n(\lg\lg n)^2 / \lg n)$ bits.

The *chunk cardinality* data structure CC: it stores the sequence of chunk cardinalities $n_1, n_2, \ldots, n_g$ in an array with fixed-size entries of $\lg b$ bits each. This takes $O(g \lg b) = O(g \lg\lg n) = O(n(\lg\lg n)^2 / \lg n)$ bits and operates in $O(1)$ time. We use implicit trees as in CW to implement CC.

The implementation of operation $\mathsf{rank}(i)$ proceeds as follows. We perform a search on CW to find the chunk c_j in which i lies. Then we search on CC to find the number x of **1**s contained in the first $j-1$ chunks of S. We get the number y of **1**s up to position i inside c_j by a table lookup on c_j itself (accessed via C and CO), since the binary encoding of c_j is at most $(\lg n)/2 + O(1)$ bits. We return $x + y$ as output, taking a total of $O(1)$ time.

2.3 Supporting select on S

It is not clear how to support both kinds of select operations using the above representations, and we take an indirect approach. Assume that S begins with a **1** and ends with a **0**; (otherwise ensure this by adding one bit before and after S at negligible cost, and adjust Proposition 1 appropriately). We describe S by two bit-vectors R_0 and R_1, defined as follows. If there are runs of **0**s of length $l_1, l_2, \ldots, l_z$ in S, then R_0 is simply $\mathbf{0}^{l_1-1}\mathbf{1}\mathbf{0}^{l_2-1}\mathbf{1}\ldots\mathbf{0}^{l_z-1}\mathbf{1}$. R_1 is defined analogously, using the runs of **1**s. See Fig. 1 for an example.

Proposition 1 ([3]). *If S has n **1**s then R_1 is of length n and has at most n **1**s. R_0 is of length $m-n$ and has at most n **1**s. Furthermore,* $\mathsf{select}_1(S,i) = \mathsf{select}_1(R0, \mathsf{rank}_1(\mathrm{R1}, i-1)) + i$ *and* $\mathsf{select}_0(S,i) = \mathsf{select}_1(R1, \mathsf{rank}_1(\mathrm{R0}, i-1) + 1) + i$, *taking* $\mathsf{select}_1(\cdot, 0) = \mathsf{rank}_1(\cdot, 0) = 0$ *on the RHS.*

We only need to support select_1 and rank_1 on R_0 and R_1. However, we will *not* store R_0 and R_1 *explicitly*, as it would take too much space. Instead, we conceptually partition R_0 and R_1 in a manner consistent with the partition of S (Lemma 1), allowing the chunks of R_0 and R_1 to be deduced from the chunks of S. Since the length of R_1 equals the number of **1**s in S, if a chunk in S contains the k-th, $(k+1)$-st, ..., $(k+l-1)$th **1**s in S, for some $l \geq 0$, then the corresponding chunk in R_1 consists of all the bits in positions $k, k+1, \ldots, k+l-1$ of R_1. (The chunk in R_1 may be of cardinality zero.) Similar observations apply for R_0. Now, the chunks of S and their homologous chunks in R_0 and R_1 do not cross block boundaries. We state a few facts about R_1 that apply to R_0 unless noted otherwise. R_1 consists of at most $g = n/s + m/b = O(n \lg\lg n / \lg n)$ chunks, each of length b, and contains at most s **1**s. (R_0 can have $s+1$ **1**s.) The

representation of a chunk x in S, together with the bit that immediately follows x in S, implicitly determines the corresponding chunk in R_1.

Operation rank on R_1 and R_0. These proceed exactly as rank on S (Section 2.2), but we cannot store R_1 and R_0 explicitly. We recover the jth chunk of R_1 and R_0 by using CO, a table lookup on its corresponding chunk c_j (stored using at most $(\lg n)/2 + O(1)$ bits, and the first bit of c_{j+1} (if any). This process takes $O(1)$ time, and the lookup table requires $O(\sqrt{n} \lg n)$ bits. We also build data structures to support rank on R_1 (R_0), similar to those on S, namely, chunk width $CW(R_1)$, stored in $O(n(\lg \lg n)^2/\lg n)$ bits, and chunk cardinality $CC(R_1)$, stored in $O(n(\lg \lg n)^2/\lg n)$ bits. Since we can retrieve any chunk of R_1 in $O(1)$ time, the implementation of rank on R_1 and R_0 is a straightforward generalization of what we discussed in Section 2.2. Time complexity is still $O(1)$.

Operation select_1 on R_1 and R_0. We describe how to perform select_1 on R_1. (R_0 is similar.) The chunks of R_1 and R_0 are obtained implicitly in $O(1)$ time, so we just need the following additional data structure.

The *ones distribution* data structure $OD(R_1)$: it is a bit-vector with g **0**s and at most n **1**s; in effect, a unary encoding of the distribution of **1**s of R_1 to the chunks of R_1. Each of the **0**s delimits a chunk boundary and the non-negative number of **1**s between consecutive **0**s indicates the cardinality (number of **1**s) of the corresponding chunk. We store the bit-vector using Theorem 1, taking $O(g \lg(n/g)) = O(n(\lg \lg n)^2/\lg n)$ bits.

The implementation of $\mathsf{select}_1(i)$ on R_1 uses $OD(R_1)$. We compute $r = \mathsf{select}_1(i)$ and $j = \mathsf{rank}_0(r) + 1$ (both on $OD(R_1)$). This tells us that the jth chunk in R_1 contains the ith **1** of R_1. Using $CC(R_1)$, we find a non-negative number x, which is the number of **1**s in R_1 up to the start of chunk c_j. We then look up CO to access chunk c_j (and the first bit of c_{j+1}), and the lookup table to deduce the jth chunk of R_1. At this point, using another table, we can select the $(i - x)$th **1** that appear in the jth chunk of R_1. This displacement is added to the prefix sum of the widths of the first $j - 1$ chunks in R_1 (computed using $CW(R_1)$), and the result is given as the output of $\mathsf{select}_1(i)$ on R_1. This takes $O(1)$ time. This discussion completes the proof of Theorem 2.

3 Compressed Dense FIDs

3.1 Informative Encodings

Recall that informative encodings of an object use space very close to the information-theoretic bound, but allow some properties of the object to be deduced by looking at a few consecutive bits of the encoding. We now show the existence of one kind of informative encoding:

Lemma 2. *Given a finite universe U, and a partition of U into t sets $C_1, \ldots, C_t$. For any probability distribution $\mathcal{P}$ over U such that for all i, and all $x, y \in C_i$, $\mathcal{P}(x) = \mathcal{P}(y) > 0$, we can encode any $x \in U$ such that:*

– *the encoding takes at most* $\lceil \lg(1/\mathcal{P}(x)) \rceil + 2$ *bits, and*
– *by inspecting* $\lg t + O(1)$ *consecutive bits of the encoding of* x*, we can determine the index* i *such that* $x \in C_i$*.*

Proof. Consider the alphabet $\Sigma = \{\text{“1”}, \text{“2”}, \ldots, \text{“}t\text{”}\}$. We construct a prefix code ϕ on Σ. If $x \in C_i$, the encoding of x comprises $\phi(\text{“}i\text{”})$ followed by an encoding of x as an integer in the range $1..|C_i|$, using $\lceil \lg |C_i| \rceil$ bits. Clearly, this is a valid encoding of x; we next show the existence of a suitable prefix code ϕ.

Let p_i be the (common) probability of all elements in C_i, and take $l_i = \lceil \lg(1/p_i) - \lg |C_i| \rceil$. We require that the length of $\phi(\text{“}i\text{”})$ is at most l_i, so that the total length of the encoding of x is $l_i + \lceil \lg |C_i| \rceil + 1 = \lceil \lg(1/p_i) \rceil + 1 = \lceil \lg(1/\mathcal{P}(x)) \rceil + 1$ bits. Now observe that $\sum_{i=1}^{t} 2^{-l_i} \leq \sum_{i=1}^{t} p_i \cdot |C_i| = 1$, the latter equality holding since $\mathcal{P}$ is a probability distribution. Thus, by the Kraft-McMillan inequality, a prefix code ϕ' exists where the code for “i” has length at most l_i, and indeed ϕ' can be constructed by a simple greedy algorithm given the lengths l_i.

To limit on the length of the code of an individual symbol, we create a new prefix code ϕ from ϕ' as follows. Whenever ϕ' assigns a code that is longer than $\lceil \lg t \rceil$ bits to a symbol, ϕ encodes the symbol using a fixed-width code of length $\lceil \lg t \rceil$ bits, otherwise ϕ uses the code assigned by ϕ' to the symbol. To distinguish between the two cases, each code in ϕ is prefixed with an additional bit to specify whether the symbol is encoded using ϕ' or the fixed-width code. Using ϕ as our prefix code, the length of the encoding of x increases to at most $\lceil \lg(1/\mathcal{P}(x)) \rceil + 2$ bits, but allows the index i to be decoded by reading at most $\lceil \lg t \rceil + 1$ bits.

We now illustrate a use of this lemma. Suppose we consider U to be all subsets of $[k]$ and, for $i = 0, \ldots, k$, we let $C_i = \{S \in U || S| = i\}$. For all $S \subseteq [k]$, we take $\mathcal{P}(S) = 2^{-k}$ (i.e. the uniform distribution over subsets of $[k]$). Then we get an encoding of a given $S \subseteq [k]$ that uses at most $k + 2$ bits, such that $|S|$ can be determined by reading $\lg(k+1) + O(1)$ contiguous bits of the encoding of S. We now show a more complex application of Lemma 2:

Corollary 1. *Let* $k, t > 0$ *be integers, and* $p, q > 0$ *be reals with* $p + q = 1$*. Given a sequence of* kt *bits, viewed as* t *contiguous subsequences of* k *bits each, there is an encoding of this sequence using is at most* $\lceil r \lg(1/p) + (kt - r) \lg(1/q) \rceil + 4$ *bits, which is a concatenation of the following parts:*

1. *a prefix code of at most* $\lg(kt) + O(1)$ *bits that specifies the number* r *of* **1** *bits in the sequence.*
2. *an encoding of the positions of the* **1** *bits that occupies at most* $B(kt, r) + O(1)$ *bits, itself comprising: (a) a prefix code of* $t \lg(kt) + O(1)$ *bits, that specifies the number of* **1** *bits in each subsequence, and (b) an encoding of the positions of the* **1** *bits in each subsequence.*

3.2 Compressed Dense FIDs Via Informative Encodings

We first show the following generalization of [12, Theorem 3] (proof omitted) before proving the main result of this section, Theorem 3:

Lemma 3. *Let $t > 0$ be an integer and let $\bar{u} = (u_1, \ldots, u_t)$ be a sequence of positive integers. Given t and $\bar{u}$, and a positive integer parameter $z < t$, one can represent any sequence of positive integers $\bar{x} = (x_1, \ldots, x_t)$, where $1 \le x_i \le u_i$, using $\sum_{i=1}^{t} \lg u_i + O(1 + t/z)$ bits, in such a way that x_i can be accessed in $O(1)$ time, for any i, on a RAM with word size $O(t \lg z + \lg \max_i\{u_i\})$ bits, using precomputed tables of size $O(t(z^t \lg z + \lg(\sum_{i=1}^{t} \lg u_i)))$ bits that depend upon $\bar{u}$, t and z, but not upon $\bar{x}$.*

Theorem 3. *We can store a FID using $B(m, n) + O(m \lg\lg m/(\lg m)^2)$ bits.*

Proof. We divide the input bit sequence into *blocks* of $k = \lceil (\lg m)/2 \rceil$ bits each, *superblocks* comprising $t = \lceil \alpha \lg m / \lg\lg m \rceil$ blocks each for sufficiently small $\alpha > 0$, and *megablocks* comprising t superblocks each. Letting $p = n/m$ and $q = 1-p$, we aim to represent a megablock with r **1**s in $r \lg(1/p) + (kt^2 - r)\lg(1/q) + O(t)$ bits, so that rank and select within a megablock can be supported in $O(1)$ time; the sizes of the megablock representations add up to $n \lg(1/p) + (m-n)\lg(1/p) + O(m/(kt)) = B(m, n) + O(m \lg\lg m/(\lg m)^2)$ bits.

A superblock is encoded along the lines described in Corollary 1, using Lemma 3 to implement item 2(b), with a few additional details. Specifically, let R_i denote the number of **1**s in the i-th superblock. The prefix codes for the R_is (item (1) from Corollary 1) are concatenated at the start of the megablock, forming a *megablock overview*. Since each prefix code is $O(\lg(kt)) = O(\lg\lg m)$ bits, the megablock overview takes at most $O(t \lg\lg m) \le (\lg m)/8$ bits if α is chosen sufficiently small. The megablock overview is followed by the encoding of the superblocks, where the i-th superblock is stored in $B(kt, R_i) + O(1)$ bits, for $i = 1, \ldots, t$. If a superblock's encoding is shorter than $B(kt, R_i) + O(1)$ bits, it is padded out to this length: this ensures that table lookup on the megablock overview allows us to determine the offset where the encoding of the ith superblock begins. The megablock overview also allows (via table lookup) a rank or select in a megablock to be reduced to that in a superblock.

Consider a particular superblock and denote its blocks by $b_1, \ldots, b_t$, and let the number of **1**s in b_i be r_i; let $r = \sum_{i=1}^{t} r_i$. We use Lemma 3 to store the encodings of the blocks, with parameters t as here, and $z = t$. The size of the tables for constant-time decoding of an individual block is easily seen to be $O(m^\epsilon)$ bits, for any $\epsilon > 0$, by choosing α small enough, and these tables can be re-used by all superblocks that have exactly the same distribution of **1**s in their blocks. The prefix code for the tuple $(r_1, \ldots, r_t)$ is $\alpha \lg m + O(1)$ bits long, and it can be decoded using tables of size $O(m^{\epsilon'})$ bits, for any $\epsilon' > 0$, by choosing α small enough. This also shows that the number of distinct tuples $(r_1, \ldots, r_t)$ is $O(m^{\epsilon'})$, and so the tables across all applications of Lemma 3 have size $O(m^{\epsilon+\epsilon'})$.

We now complete the proof. Let M_i denote the number of **1**s in the ith megablock. We create bit sequences $D_0 = \mathbf{0}^{kt^2-M_1}\mathbf{1}\mathbf{0}^{kt^2-M_2}\mathbf{1}\ldots$ and $D_1 = \mathbf{0}^{M_1}\mathbf{1}\mathbf{0}^{M_2}\mathbf{1}\ldots$, and store them as FIDs using Theorem 2. Note that both D_0 and D_1 have $m/(kt^2)$ **1**s (equal to the number of megablocks) and at most m **0**s, and thus have a sufficiently large number of **1**s relative to their total size that Theorem 2 applies. Further, note that the FIDs occupy $O(m(\lg\lg m/\lg m)^3)$ bits

and so have negligible size. It is also easy to verify that using rank and select operations on D_0 and D_1 one can reduce rank and select operations on the input bit sequence to operations on the megablocks.

4 Representing Balanced Parenthesis Sequences

Given a balanced parenthesis sequence (BP, for short) of length m, we want to represent it space-efficiently to support the following operations, for a position $1 \leq i \leq m$: findopen(i) (findclose(i)) returns the location of the matching parenthesis for an opening (closing) parenthesis at the location i; enclose(i): returns the open parenthesis of the pair that most tightly encloses the location i.

4.1 Lower Bound

We show a lower bound on the size of the index required to support findclose on a given BP, S, that is represented in memory in its verbatim form. On a query findclose(i), our algorithm performs at most τ bit probes on S (adaptively), any number of bit probes on an index I of size r, and also allowed unlimited computation.

Theorem 4. *Given a balanced parenthesis sequence S of length m, any index that supports* findclose *queries by making τ bit probes to S requires an additional index of size $r = \Omega(m \lg \tau/\tau)$ bits.*

Proof. We use techniques similar to [6]. Let us consider a set $\mathcal{B}$ of balanced parenthesis strings S of the following form: (i) S starts with hp opening parentheses for some parameters h and p, (ii) the rest of S is divided into $2p$ *blocks* of width k, $k = (m - hp)/(2p)$; and (iii) the matching parenthesis $p_i =$ findclose($ih + 1$) is located in the $2(p - i)$-th block for $0 < i < p$, and findclose(1) $= n$.

$$S = \underbrace{(\overbrace{(\ldots(}^{hp}\underbrace{*\,*\ldots*}_{k}\underbrace{*\,*\ldots}_{k})\ldots*\ldots\underbrace{*\,*\ldots}_{k})\ldots*\underbrace{*\,*\ldots}_{k}*\underbrace{*\,*\ldots*}_{k})}_{2p \text{ blocks}}$$

We start by showing that $|\mathcal{B}| = \Omega(2^{m-hp-\Theta(p)})$. First, divide S into *chunks*, so that the first chunk starts at the position $hp + 1$ and ends at the position findclose($i(p - 1) + 1$), and the $(p + 1 - i)$-th chunk starts at the position findclose($ih + 1$) $+ 1$ and ends at the position findclose($(i - 1)h + 1$) for i, $1 \leq i \leq p - 1$. Let $m_1, m_2, \ldots, m_p$ denote the width of these chunks, $\sum m_i = m - hp$, and note that $k \leq m_i \leq 3k$ for all i, $1 \leq i \leq p$. We use Bertrand's ballot theorem to bound the number of sequences with fixed width of chunks. This theorem shows that "If in an election where one candidate receives x votes and the other y votes with $x \geq y$, the probability that the first candidate be strictly ahead of the second candidate throughout the count is $(x-y)/(x+y)$". We can look at the i-th chunk from right to left (i.e. backward) as a sequence of votes: closing parenthesis is a $+1$ vote, and open parenthesis is a -1 vote. The sum of the votes is h at the end, and at each moment of

time, it is strictly positive. Thus, the number of possible blocks of length m_i is $\frac{h}{m_i}\binom{m_i}{(m_i+h)/2} = \Omega\left(\frac{h}{m_i}\binom{m_i}{m_i/2}\right) = \Omega\left(\frac{h}{m_i}\frac{2^{m_i}}{\sqrt{m_i}}\right)$, if $h = O(\sqrt{m_i})$. Taking product over all i, we obtain $\Omega\left(2^{m-hp-\Theta(p)}\frac{h^p}{k^{3p/2}}\right)$. We can choose the position p_i within the $2(p-i)$-th block arbitrarily respecting the parity constraint (the width of any block should be the same parity as h), so that there are at least $\Omega((k/2)^p)$ possible combinations of blocks' boundaries. If $h = \sqrt{k}$, then $|\mathcal{B}| \geq 2^{m-ph-\Theta(p)}$.

The *choices tree* is a binary tree, the nodes of the first r levels are labeled with "$I[d]$=?" where d is the depth of the node, and the rest of nodes are labeled with "$S[l]$=?" for some l, $1 \leq l \leq m$. The edges are labeled with 0 or 1. The first r levels of this tree correspond to the index. At each node at depth r of the tree constructed so far, we attach the decision tree of the algorithm on the query findclose(1). At each leaf the previously constructed tree, we attach the decision tree for findclose($h+1$) and so on. If on the same computation path we discover two nodes of the form "$S[l]$=?" for the same value of l, we delete all of them except the topmost. We also delete the nodes that probe the first hp locations. We pad the tree by probing arbitrary locations, so that all the leaves are at the same level $r + p\tau$.

Let us fix a leaf x, we call a string S *compatible* with x if: (1) first r nodes on the root to the leaf path correspond to the index for S; and (2) the remaining nodes on the root to the leaf path correspond to the choices made by the computation described above performed on S. Let $C(x)$ be the set of compatible strings for the leaf x. Let u_i be the number of unprobed locations in the i-th chunk, and v_i be the number of unprobed locations that correspond to opening parentheses, v_i equals to $(m_i - h)/2$ minus the number of revealed "(" on the computation path to x. Both values u_i and v_i depend on the leaf x only, but not on the choice of string $S \in C(x)$. Thus,

$$C(x) \leq 2^U \frac{\binom{u_1}{v_1}}{2^{u_1}}\frac{\binom{u_2}{v_2}}{2^{u_2}}\cdots\frac{\binom{u_p}{v_p}}{2^{u_p}} \leq \frac{\binom{u_1}{u_1/2}}{2^{u_1}}\frac{\binom{u_2}{u_2/2}}{2^{u_2}}\cdots\frac{\binom{u_p}{u_p/2}}{2^{u_p}} = O\left(\frac{2^U}{\sqrt{u_1}\sqrt{u_2}\cdots\sqrt{u_p}}\right)$$

where $U = \sum u_i = m - hp - p\tau$ is the total number of unprobed locations. Let $p = m/(6\tau)$, so that $k = 3\tau - o(\tau)$ (since $h = O(\sqrt{\tau})$). We say that the i-th chunk is *determined* if $u_i < \tau$, i.e. at least 2τ locations are probed, the number of such chunks is at most $p/2$. Thus, the product in the denominator is at least $\tau^{m/(12\tau)}$. The total number of leaves in the choices tree is $2^r 2^{p\tau}$, so that $2^{m-hp-\Theta(p)} \leq |\mathcal{B}| \leq 2^r 2^{p\tau} 2^{m-hp-p\tau}\tau^{-m/(12\tau)}$, and hence $r \geq (m \lg \tau)/(12\tau) - \Theta(m/\tau)$.

4.2 Upper Bound

Theorem 5. *A BP of length m can be encoded in $m + O(m \lg\lg m/(\lg m)^2)$ bits to support* findclose, findopen *and* enclose *in $O(1)$ time.*

Proof. Divide the balanced parenthesis sequence (BP) into *blocks* of size $k = (\lg m)/2$ each, *superblocks* comprising $t = O(\lg m/\lg\lg m)$ blocks each, and *megablocks* comprising t superblocks each (as in the proof of Theorem 3).

Given a parenthesis sequence x (not necessarily balanced), we define the function $\mathsf{closeexcess}(x,i) = \mathsf{rankclose}(x,i) - \mathsf{rankopen}(x,i)$ and (note that $\mathsf{rankopen}$ and $\mathsf{rankclose}$ are same as rank_0 and rank_1 as we store the BP as a bit sequence). We also define $\mathsf{maxclose}(x)$ ($\mathsf{maxopen}(x)$) as the maximum, over all prefixes of x, of the number of closing (opening) parentheses minus the number of opening (closing) parentheses. Note that $0 \leq \mathsf{maxopen}(x), \mathsf{maxclose}(x) \leq |x|$.

We partition the set of all possible blocks into sets $C_{\langle n,d,e\rangle}$ for $0 \leq n,d,e \leq k$. A block x belongs to the set $C_{\langle n,d,e\rangle}$ where $n = \mathsf{rankopen}(x,|x|)$, $d = \mathsf{maxopen}(x)$ and $e = \mathsf{maxclose}(x)$. A block x is uniquely determined by the triple $\langle n,d,e\rangle$ together with an index c in the range $1..|C_{\langle n,d,e\rangle}|$ that stores the position of x among all the blocks that belong to $C_{\langle n,d,e\rangle}$ in some canonical ordering. We also use a similar partition over all superblocks to encode them succinctly.

Using the ideas described in the proof of Theorem 3 we encode each megablock using $kt^2+O(t)$ bits that enables us to retrieve the $\langle n,d,e\rangle$ triple for any block or superblock within the megablock in constant time (using additional precomputed tables of negligible size. So the overall space used to store the encodings of all the megablocks is $(m/kt^2)(kt^2+O(t)) = m + O(m \lg\lg m/(\lg m)^2)$ bits.

Operations. For each megablock, we store $\mathsf{rankopen}$ and $\mathsf{rankclose}$ up to its beginning using Theorem 2. Since the number of megablocks is $O(m/kt^2)$ it requires $O((m/kt^2)\lg(kt^2)) = o(m/(\lg m)^2)$ bits. These values up to the beginning of a block or a superblock within a megablock can be obtained using the megablock encoding. Within a block, these values can be computed using a table lookup on the block representation. Thus we can support $\mathsf{rankopen}$ and $\mathsf{rankclose}$, and hence $\mathsf{closeexcess}$, for an arbitrary position in the sequence, in $O(1)$ time.

Findclose. We use a structure similar to the one described in Geary et al. [5], and we only mention the deviations required to reduce the space bound.

We define the *pioneer* and *far* parentheses and the *pioneer family* as in Geary et al. [5], except that we define them with respect to the megablocks instead of blocks. The original sequence is encoded as outlined above, to support $\mathsf{rankopen}$, $\mathsf{rankclose}$ and $\mathsf{closeexcess}$ operations in constant time using $m + O(m \lg\lg m/(\lg m)^2)$ bits. The *NND* storing the positions of the pioneer family (a set of size $O(m/kt^2)$ from $[m]$) takes $o(m/(\lg m)^2)$ bits by using the structure of Theorem 2. The *recursive parenthesis data structure* for the pioneer family (a BP of length $O(m/kt^2)$) also takes $o(m/(\lg m)^2)$ bits.

As all the required operations on the pioneer family can be supported in constant time, it suffices to support the following operation in constant time: given a megablock x, an index i within it and a value $0 \leq v \leq kt^2$, find the first (closing parenthesis at) position j after i within x such that $\mathsf{closeexcess}(x,j) = \mathsf{closeexcess}(x,i) + v$ (if it exists). We use precomputed tables on block representations, and on the sequences of $\langle n,d,e\rangle$ triples stored for the blocks and superblocks stored in the megablock encoding to support the above operation (and also various other operations described below, within a megablock).

Enclose. Let $\mu(i)$ denote the position of the closing parenthesis corresponding to the opening parenthesis at position i. To find $p = \mathsf{enclose}(c)$, we first check

whether p or $\mu(p)$ is in the same megablock as c. Finding $\mu(p)$ if it belongs to the same megablock as c is same as the above operation with $v = 2$. Finding p if it belongs to the same megablock as c can be done similarly.

Since operations on the pioneer family can be supported in $O(1)$ time, it suffices to show how to find the first far open parenthesis to the left of a given position q in a megablock. Within a block, this can be done in constant time by a table lookup, but it's not clear how to support it in a megablock in constant time. We further observe that it suffices to support this when the parenthesis at position q is a pioneer. In this special case, we can actually store these values (the difference between q and the answer, using $O(\lg \lg m)$ bits) for each pioneer.

References

1. Beame, P., Fich, F.E.: Optimal bounds for the predecessor problem and related problems. J. Comput. Syst. Sci. 65, 38–72 (2002)
2. Brodnik, A., Munro, J.I.: Membership in Constant Time and Almost-Minimum Space. SIAM J. Computing 28, 1627–1640 (1999)
3. Delpratt, O., Rahman, N., Raman, R.: Engineering the LOUDS succinct tree representation. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 134–145. Springer, Heidelberg (2006)
4. Gál, A., Miltersen, P.B.: The cell probe complexity of succinct data structures. Theor. Comput. Sci. 379, 405–417 (2007)
5. Geary, R.F., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theor. Comput. Sci. 368, 231–246 (2006)
6. Golynski, A.: Optimal lower bounds for rank and select indexes. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 370–381. Springer, Heidelberg (2006)
7. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th ACM-SIAM SODA, pp. 841–850. ACM Press, New York (2003)
8. Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: experiments with compressing suffix arrays and applications. In: SODA '04, pp. 636–645
9. Jacobson, G.: Space efficient static trees and graphs. In: Proc. 30th IEEE Symp. FOCS, pp. 549–554. IEEE Computer Society Press, Los Alamitos (1989)
10. Miltersen, P.B.: Lower bounds on the size of selection and rank indexes. In: Proceedings of the ACM-SIAM SODA, pp. 11–12. ACM Press, New York (2005)
11. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31, 762–776 (2001)
12. Munro, J I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 345–356. Springer, Heidelberg (2003)
13. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM J. Computing 31, 353–363 (2001)
14. Patrascu, M., Thorup, M.: Time-space trade-offs for predecessor search. In: Proc. 38th ACM STOC, pp. 232–240. ACM Press, New York (2006)
15. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries, with applications to representing k-ary trees and multisets. In: ACM-SIAM SODA '03, pp. 233–242

Compact Oracles for Approximate Distances Around Obstacles in the Plane

Mikkel Thorup

AT&T Labs—Research, Shannon Laboratory,
180 Park Avenue, Florham Park, NJ 07932, USA
mthorup@research.att.com

Abstract. Consider the Euclidean plane with arbitrary polygonal obstacles with a total of n corners. For arbitrary $\varepsilon > 0$, in $O(n(\log n)^3/\varepsilon^2)$ time, we construct an $O(n(\log n)/\varepsilon)$ space oracle that given any two points reports a $(1+\varepsilon)$ approximation of the obstacle avoiding distance in $O(1/\varepsilon^3 + (\log n)/(\varepsilon \log\log n))$ time. Increasing the oracle space to $O(n(\log n)^2/\varepsilon)$, we can further report a corresponding path in constant time per hop.

1 Introduction

As described by Hershberger and Suri [1], "the Euclidean shortest path problem is one of the oldest and best-known problems in computational geometry. Given a planar set of arbitrary polygonal obstacles with disjoint interiors, the problem is to compute a shortest path between two points avoiding all the obstacles. Due to its simple formulation and obvious applications in routing and robotics, the problem has drawn the attention of many researchers in computational geometry..." Here an obstacle avoiding path is allowed to touch obstacles, but not to cross them. This implies that a shortest path only bends at obstacle corners. With n denoting the total number of corners, Hershberger and Suri proceed to show that this off-line problem can be solved for any two points a and b in $O(n \log n)$ time, which is optimal.

The problem we consider here is to preprocess the plane with the obstacles so that we can quickly estimate the distance between any pair of points, and report a corresponding path. This kind of preprocessing for two-point queries problem has a rich history [2].

We consider the approximate version, where the distances and paths reported have stretch $1+\varepsilon$ for an arbitrarily small $\varepsilon > 0$. Formally, a distance estimator d has stretch s relative to the real distance function δ if for each pair of points a and b, we have $\delta(a,b) \leq d(a,b) \leq s\,\delta(a,b)$. Our main result is as follows.

Theorem 1. *Consider the Euclidean plane with polygonal obstacles with a total of n corners. Let $\varepsilon > 0$. In $O(n(\log n)^3/\varepsilon^2)$ time we can produce an $O(n(\log n)/\varepsilon)$ space distance oracle that estimates arbitrary obstacle avoiding distances with stretch $1+\varepsilon$ in $O(1/\varepsilon^3 + (\log n)/(\varepsilon \log\log n))$ time. Increasing the space to $O(n(\log n)^2/\varepsilon)$, we can report a corresponding path in constant time per hop.*

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 383–394, 2007.

In the above bounds, we have assumed that our algorithm is implemented in a standard imperative programming language such as C [3] and that point locations and distances are represented as integers or floating point numbers. Such representation is certainly reasonable when aiming for approximate distances. In particular, we can use distance values to compute addresses as done in bucket and radix sort, and this allows us to compute undirected single source shortest paths in linear time [4,5]. The addressing also allows us to compute nearest common ancestors in a rooted tree in constant time [6]. On a comparison-addition based pointer machine, some of the bounds would grow by a factor $O(\log n)$. Our construction works not only for the Euclidean plane but for any ℓ_p norm.

Closely related work. Based on Clarkson's [7] approximate solution to the off-line problem, Chen [8] shows that a data structure can be built in near-quadratic time and space, estimating distances with stretch $1+\varepsilon$ in logarithmic time. For now we think of ε as a small positive constant. Chen [8] also constructs a near-linear space data structure in $\tilde{O}(n^{3/2})$ time which estimates distance with stretch $6+\varepsilon$ in logarithmic time. As a later improvement, Arikati et al. [9] show that using near-linear space, we can answer rectilinear distance queries with stretch $2+\varepsilon$ in logarithmic time. For the Euclidean metric, this implies stretch $\sqrt{2}(2+\varepsilon)$. Chiang and Mitchell [10] show that we can get exact answers in logarithmic time, but their data structure uses $O(n^{11})$ space. Gudmundson et al. [11,12] show that if there is a constant bounding the ratio of direct Euclidean distances over obstacle avoiding distances, then there is a near-linear space data structure answering distance queries with stretch $1+\varepsilon$ in constant time. However, the bounded ratio rules out thin obstacles like rivers.

Varadarajan [13] pointed out a construction implicit in previous work leading to near-linear space oracle like the one claimed in Theorem 1, providing stretch $1+\varepsilon$ distances in sub-logarithmic time. However, the previous oracle construction is more complicated and slower by a factor $1/\varepsilon^4$, which matters in theory when ε is $o(1)$, and in practice if, say, $\varepsilon < 1/5$. The construction uses a $1+\varepsilon$ spanner graph over the obstacle corners claimed by Arikati et al. [9] in a final remark. They provide no details, but Zeh [14, Lemma 14.18] calculates the size of this planar spanner to be $O(n/\varepsilon^4)$. A distance oracle for this planar spanner is constructed using Klein [15] or Thorup [16]. Finally Chen [8] has shown how to use a corner distance oracle to get obstacle avoiding distances between arbitrary pairs of points.

The technical contribution of this paper is to bypass the spanner construction above and instead modify the planar graph distance oracle construction from [16] to work directly with obstacles in the Euclidean plane. Instead of working with the planar spanner with $O(n/\varepsilon^4)$ vertices and edges, we work with Clarkson's [7] cone graph whose vertices are the n obstacle corners connected by $O(n/\varepsilon)$ edges. The extra edges do not affect the running time because the algorithm internally works with even more auxiliary edges. The fewer vertices gain us a factor $1/\varepsilon^4$ in construction time. The cone graph data structure is used internally in the previous construction, so it does not add new complexity. The cone graph is highly non-planar, but in a way that can be cut efficiently by the planar path

separators from [16]. For contrast, the traditional planar vertex separators of Lipton and Tarjan [17] cannot cut the cone graph.

2 Borrowed Framework and Techniques

Instead of the parameter ε, we will use an inverse integer parameter k, and get a stretch of $1 + O(1/k)$. We will use a technique of Clarkson [7] the off-line approximate shortest path problem among polygonal obstacles.

For any point a, the view is divided into k cones. Cone $i \in [k]$ covers directions between $360°(i-1/2)/k$ and $360°(i+1/2)/k$. Here, $0°$ represents some fixed base direction. For any cone i, we want to know the closest visible obstacle corner v. Here closeness is measured as projected on the central cone direction $360°i/k$. Unless otherwise stated, we will always have this understanding of closeness to a within a given cone i from a. Clarkson [7] presented a data structure for this problem called a *cone Voronoi diagram*. The cone Voronoi diagram is constructed in $O(nk \log n)$ time and $O(nk)$ space, and it supports queries in $O(\log n)$ time.

Clarkson [7] constructs a cone graph G whose vertices are the obstacle corners. For each corner u, he uses the cone Voronoi diagram to find the nearest corner $v_i \neq u$ in each of the k cones, and add the *cone edge* (u, v_i). This cone edge embeds into the straight line from u to v_i in the plane. The line does not cross any obstacle, so a path in G embeds into an obstacle avoiding path. Clarkson proves that a shortest path in G has stretch $1 + O(1/k)$ compared with shortest obstacle avoiding path in the plane. The cone graph G may be very far from planar in that the embedding of cone edges may cross arbitrarily.

Note that a cone edge (u, v_i) is asymmetric in the sense that u may not be the corner nearest v in the reverse cone from v. We will use the cone graph as an undirected graph, but remember the vertex u a cone edge is generated from.

To solve the shortest path problem approximately, Clarkson adds a and b as single point obstacle corners, and construct the above cone graph. The graph is constructed in $O(kn \log n)$ time. It has $n + 2$ nodes and at most $k(n + 2)$ edges. The shortest graph path from a to b is computed in $O(kn + n \log n)$ time. We shall need the following from Clarkson's work:

Lemma 1 (Clarkson[1]). *Given the plane with polygonal obstacles with a total of n corners, we can construct a cone Voronoi diagram in $O(nk \log n)$ time and $O(nk)$ space. Subsequently, for any cone i from a point a, we find a closest visible corner, if any, in $O(\log n / \log \log n)$ time. In particular, we can build the cone graph in $O(nk \log n)$ time. The vertices of the cone graph are the n obstacle corners connected by $O(nk)$ edges, and it represents obstacle avoiding distances between the corners with stretch $1 + O(1/k)$.*

Chen [8] has shown how to modify Clarkson's approach for the off-line shortest path problem into a preprocessing for two point queries. The preprocessing does not know the query points, but computes all-pairs shortest paths in Clarkson's

[1] We base our time bounds on the recent point location of Chan and Patrascu [18,19] which takes $O(\log n / \log \log n)$ time as opposed to the classic $O(\log n)$ time.

cone graph over all the original obstacle corners. This takes $\tilde{O}(kn^2)$ time. The resulting distance matrix is used as a stretch $1 + O(1/k)$ oracle for obstacle avoiding distances between corners.

Chen uses the conical Voronoi diagrams to construct cone edges from query points a and b to the obstacle corners. This takes $O(k \log n)$ time. For each cone edge (a, u) from a and (v, b) to b, he uses the corner oracle to estimate the distances from u to v, and add this to the Euclidean distances from a to u and from v to b. The smallest such value is a stretch $1 + O(1/k)$ estimate of the shortest path from a to b that touches some obstacle corner on the way. Computing all these combinations takes $O(k^2)$ time.

We have now simulated Clarkson's off-line algorithm except for the case with a direct cone edge between a to b, in which case the answer to the distance query should be the Euclidean distance from a to b. Chen augments the cone Voronoi diagram from the corners with extra information so as to check for this special case in $O(k \log n)$ time. Thanks to this fix of Chen, we can forget about this special case. However, we shall need some details from Chen's solution in our own construction, so we state here the relevant lemmas. First we have

Lemma 2 (Chen). *Consider a set of polygonal obstacles plus two arbitrary points a and b. Consider the cone i from a that contains b, that is, the direction to b is between $360°(i - 1/2)/k$ and $360°(i + 1/2)/k$. Suppose b is as close to a as any visible obstacle corner in cone i. If b is not visible from a then the first obstacle hit by the line from a to b is an obstacle segment (u, v) that is also the first obstacle hit by one of the cone boundary lines from a in directions $360°(i - 1/2)/k$ and $360°(i + 1/2)/k$.*

We already have Clarkson's cone Voronoi diagram from Lemma 1 which tells us the closest visible corner u in cone i from a, and if u is closer in the central cone direction than b, then we know that a cone edge from a to b is not needed. Hence we can assume that b is as close as any visible corner in cone i. Now to check if a cone edge from a to b is appropriate, using Lemma 2 we consider each directions $360°(i - 1/2)/k$ and $360°(i + 1/2)/k$, and find the first obstacle segment (u, v) hit, if any, and check if it blocks b from the view of a. If no such blocking is found, we know that a sees b, and then we return $\|a - b\|_2$ as the exact distance from a to b. To find the first obstacle hit in a cone boundary direction $360°(i + 1/2)/k$, Chen uses the preprocessing of the following lemma.

Lemma 3 (Chen). *For a given direction, we can preprocess the plane with polygonal obstacles with a total of n corners in $O(n \log n)$ time and $O(n)$ space so that for any query point a, we can identify the first obstacle point hit in the given direction in $O(\log n / \log \log n)$ time. Applying this preprocessing for each of the k cone boundary directions $360°(i + 1/2)/k$, the preprocessing takes $O(nk \log n)$ time and $O(kn)$ space while the query time along any one of these directions remain $O(\log n / \log \log n)$.*

Using Lemma 2 and Lemma 3, we can check if a cone edge from a to b is appropriate in $O(\log n / \log \log n)$ time. We note that Chen's construction can be used in conjunction with any distance oracle over the obstacle corners.

Lemma 4 (Chen). *Suppose we have an oracle that estimates the obstacle avoiding distance between obstacle corners with stretch s in time t. With an additional preprocessing $O(nk \log n)$ time and $O(kn)$ space, we can estimate the distance between any pair of points in $O(k^2 t + k \log n / \log\log n)$ time and stretch $\max\{s, 1 + 1/k\}$.*

The corner oracle based on an all pairs shortest path computation over Clarkson's cone graph has $s = 1 + O(1/k)$ and $t = O(1)$, which is very good, but the preprocessing time and space is near-quadratic. Chen presents a more economic oracle for corner distances. He constructs it in $\tilde{O}(n^{3/2})$ and it uses only near-linear space. However, the corner distance stretch is 6, which then becomes the stretch of his distance estimates for arbitrary two point queries. This paper is also about a more efficient distance oracle for corners, referring to Lemma 4 for distances between arbitrary points.

3 An Approximate Distance Oracle for Obstacle Corners

We will now show how to construct an oracle providing stretch $1 + O(1/k)$ distances between the corners of the obstacles. This is a modification of the technique of Klein and Thorup [15,16] to construct approximate distance oracles for the vertices of an undirected planar graph. In this section we show the existence of such an oracle, and pay only little attention to the construction time. This part is very similar to the planar graph construction by Thorup [16], yet our presentation is focused on the geometric case. The efficient construction in the next section is more technically interesting.

Distance notation. As a general notation, we shall use $\delta(a, X_1, ..X_j, b)$ to denote the distance from a to b passing $X_1, ..., X_j$. Here each X_i represents a portion of the plane, e.g., X_i could be a single point, a set of discrete points, or a curve. We want the shortest length of a path from a to b that passes points $p_1, ..., p_j$ in this order with $p_i \in X_i$. For example, if V is the set of obstacle corners, then $\delta(a, V, b)$ denotes the shortest distance via some corner.

Distances in a domain X are denoted δ_X. If the domain is understood, the subscript X may be omitted. Currently, δ denotes distances in the plane without obstacles. Now take Clarkson's cone edge graph G from Lemma 1. The fact that it represents distances between corner obstacles u and v with stretch $1 + O(1/k)$ can be stated as $\delta(u, v) \le \delta_{G_k}(u, v) \le (1 + O(1/k))\, \delta(u, v)$.

Separator paths. As a first step we pick an arbitrary corner from which we construct an exact shortest path tree T spanning all the obstacle corners. Hershberger and Suri [1] have shown that this can be done in $O(n \log n)$ time and space. Next, starting from the polygonal lines of the obstacles and the tree T, we triangulate the plane, both inside and outside the obstacles. We assume that our plane has a polygonal frame so the triangulation can be done with straight line segments inside the frame. The exterior region outside the frame may not be used by any shortest path and is triangulated with curved edges.

The triangulated plane is now a planar graph with a spanning tree T. As described in [16], in $O(n)$ time, we can find a triangle Δ so that if we look at the region of the fundamental cycle induced by any side (u, v) of Δ, then this cycle contains less than half the triangles. Here, the fundamental cycle consists of (u, v) together with the unique path in T between u and v. As a degenerate case, we note that if $(v, w) \in T$, the inside is empty.

We now have a separator S of the plane which is the union of at most 6 shortest paths; namely the at most three paths in T from the triangle corners plus any triangle side which is not inside an obstacle or in the exterior region. We refer to these paths as *separator paths*. Working with planar graphs, Thorup [16] did not need to include the triangle sides in his separators.

Connections via a shortest path. We will now show how to represent approximate distances via a given shortest path Q. We are going to do this for each of the at most 6 shortest paths in the above separator. For corners u and v, we want to approximate $\delta(u, Q, v)$. So far, the representation is only existential.

For each vertex u, we want a set of *connections* $C(u, Q)$ which are auxiliary weighted edges from u to points q_i in Q. The weight of (u, q_i) is $\delta(u, q_i)$. These connections have stretch s if for every point q in Q, the distance from u in $C(u, Q) \cup Q$ is at most s times longer than the original distance. Recall that Q is a shortest path. Hence, if we have a connection $(u, q_i) \in C(u, Q)$ and q is any point in Q, then $\delta_{Q \cup C(u,Q)}(u, q) \leq \delta(u, q_i, q)$.

Lemma 5. *There is a set $C(u, Q)$ of $4k$ connections from u to Q with stretch $1 + 1/k$.*

Proof. Let q_0 be the vertex in Q nearest u, that is, $\delta(u, q_0) = \delta(u, Q)$. We will now move from q_0 towards one end t of Q, identifying connection points q_i. Having identified q_i, $i > 0$, we pick q_i as the first subsequent point in Q such that $(1 + 1/k)\delta(u, q_i) < \delta(u, q_{i-1}, q_i)$.

We claim that this process results in at most $2k$ connections. The point is that $\delta(u, q_i) < \delta(u, q_{i-1}, q_i) - \delta(u, q_i)/k \leq \delta(u, q_{i-1}, q_i) - \delta(u, q_0)/k$. Hence $\delta(u, q_i) < \delta(u, q_0, q_i) - i\,\delta(u, q_0)/k$, so $i\,\delta(u, q_0)/k < \delta(u, q_0) + (\delta(q_0, q_i) - \delta(u, q_i)) \leq 2\,\delta(u, q_0)$. Thus $i < 2k$. The same procedure is applied to the other end-point of Q, so we end up with at most $1 + 2(2k - 1) \leq 4k - 1$ connections. □

Representing connections for efficient queries. We will now show how to represent connections to Q so that we can quickly estimate distances via these connections. We chose one end of Q as the starting point, and let $Q[x]$ denote the point at distance x from this starting point. We call x the *offset* of $Q[x]$ in Q. For each connection $(u, q_i) \in C(u, Q)$, let x_i be the offset of q_i. We represent this connection by the pair (x_i, y_i) where $y_i = \delta(u, q_i)$. Now, for any point $q = Q[x]$ in Q, we have $\delta(u, q_i, q) = y_i + |x - x_i|$. The connections in $C(u, Q)$ are always sorted according to the offset.

Now, consider two corners u and v. Suppose we have a connection $(x, y) \in C(u, Q)$ from u to Q and a connection $(x', y') \in C(v, Q)$ from v to Q. Then $\delta(u, Q[x], Q[x'], v) = y + |x - x'| + y'$. This way we use connections from u and v to Q to get distances from u to v via Q.

To estimate $\delta(u, Q, v)$ from $C(u, Q)$ and $C(v, Q)$ we first merge $C(u, Q)$ and $C(v, Q)$. In the resulting sorted list we look for neighbors $(x, y) \in C(u, Q)$ and $(x', y') \in C(v, Q)$. This neighboring pair corresponds to the distance $y + |x - x'| + y' = \delta(u, Q[x], Q[x'], v)$, and we return the minimal such distance via neighboring connections from u and v. We denote this estimate $d(u, Q, v)$.

Lemma 6. *If $C(u, Q)$ and $C(v, Q)$ provide connections from u and v to Q with stretch s, then $d(u, Q, v)$ estimates $\delta(u, Q, v)$ with stretch s in $O(|C(u, Q)| + |C(v, Q)|)$ time.*

Proof. The query time is trivially linear. Now consider the shortest path P from u to v intersecting Q in some point $Q[z]$. Since $C(u, Q)$ has stretch s, it has a connection (x, y) such that $\delta(u, Q[x], Q[z]) = y + |x - z| \leq s\,\delta(v, Q[z])$. Similarly, we have a connection $(x', y') \in C(v, Q)$ such that $\delta(v, Q[x'], Q[z']) = y' + |x' - z| \leq s\,\delta(v, Q[z])$. Hence $y + |x - x'| + y' \leq y + |x - z| + |x' - z| + y' \leq s\,\delta(u, Q[z]) + s\,\delta(Q[z], v) = s\,\delta(u, Q, v)$. Now, it may that the above (x, y) and (x', y') are not neighbors. However, let (x, y) and (x', y') be picked as close as possible yet producing the minimal distance. Then they must be neighbors; for otherwise there is a connection (x^*, y^*) between them. By symmetry, we can assume $(x^*, y^*) \in C(v, Q)$. Then $\delta(u, Q[x], Q[x'], v) = \delta(u, Q[x], Q[x^*], Q[x'], v) \geq \delta(u, Q[x], Q[x^*], v)$. Hence we get a no worse distance using (x^*, y^*) instead of (x', y'), contradicting the choice of (x', y'). Thus we conclude that (x, y) and (x', y') are neighbors, providing the distance with the desired stretch. □

A recursive distance oracle. We are now ready to describe our distance oracle for the corner recursively. We have a separator S consisting of at most 6 shortest paths. Using Lemma 5, for each vertex v and separator path $Q \in S$, we get a set of $O(k)$ connections with stretch $1 + 1/k$. The total space is now $O(kn)$.

Having represented the distances via the separator, we cut the plane along the 6 separator paths, and recurse on each region as an independent subproblem. This cutting may split corners and edges into multiple copies. Including the parts of T along the boundary of a piece, each piece inherits a shortest path tree and a triangulation. The size of a subproblem is measured by the number of triangles. Each time we divide into subproblems, each subproblem has at most half as many triangles, so the recursion depth is $O(\log n)$. We continue recursing until the subproblems are individual triangles, and we store the recursion tree $\mathcal{R}$ as part of our representation.

A corner u is only a participant in a subproblem Φ if it belongs to the region of the subproblem and it is not part of a separator on a higher levels. Then u can only participate in one subproblem on each level. It is only from participating corners that we store connections to the at most 6 separator paths Q of Φ. Hence the total space for these connections is $O(nk \log n)$. For each corner u we store the lowest subproblem Ψ_u that u participates in. Then u is part of the separator of this subproblem.

Now, how do we answer distance queries between vertices u and v? From Ψ_u and Ψ_v, we go up the recursion tree finding the $O(\log n)$ common ancestor problems Φ that both u and v participate in. For each common subproblem Φ,

we compute the estimate $d_\Phi(u, Q, v)$ via each separator path Q in $O(k)$ time. The smallest estimate over all common subproblems is returned. The total query time is then $O(k \log n)$.

To see that the stretch is $1+1/k$, consider a shortest path P from u to v. As we perform the recursion, there will be a first subproblem Φ with a separator path Q intersecting P. Since P was not intersected by a separator path on a higher level, we know that u and v both participate in this subproblem and that P is in Φ. Hence we have $\delta(u, v) = |P| = \delta_\Phi(u, v) = \delta_\Phi(u, Q, v)$. and then our connections in Φ from u and v to Q gives us an estimate $d_\Phi(u, Q, v) \leq (1+1/k)\, \delta_\Phi(u, Q, v) = (1 + 1/k)\delta(u, v)$.

4 Efficient Construction

We will now describe an efficient construction of the connections to separator paths. Technically, this is the most novel part of the paper. First we construct the recursive separator hierarchy. So far we have not made any connections. The construction time for the recursive hierarchy $\mathcal{R}$ is $O(n \log n)$.

We shall use Clarkson's cone edge graph G from Lemma 1 which represent distances between obstacle corners with stretch $1 + O(1/k)$. To construct the graph, for each corner u and each of k cones C_u from u, we added a straight line cone edge from u to the nearest corner v in C_u, if any. The result is not a planar graph as the straight line edges may cross. However, the straight lines do not cross any obstacle or the external boundary. Hence each path P in G is embedded into an obstacle avoiding path in the plane.

Our overall goal is to provide corner distance estimates $d(u, v)$ so that

$$\delta(u, v) \leq d(u, v) \leq (1 + 1/k)\delta_G(u, v). \tag{1}$$

Note that $d(u, v)$ is only lower bounded by the real distance in the plane with obstacle. Since $\delta_G(u, v) \leq (1 + O(1/k))\delta(u, v)$, we have that (1) implies that d has stretch $1 + O(1/k)$ compared with the real distances.

We are going to construct connections to a separator path $Q \in S$ which is a shortest obstacle avoiding path in the plane. The precise goal is that if there is a u-v path P in G whose embedding touch Q, the connections via Q should provide a distance estimate bounded by $(1 + 1/k)|P|$.

Our first step is to construct a new graph G^Q from G so as to encode the relationship between the graph G and the plane separator path Q. First we add all segments of Q as graph edges. Next we consider all intersections between cone edges (u, v) and segments (f, g) of the separator path Q. If they intersect in a point p, we make p a new auxiliary vertex, and replace (u, v) with (u, p) and (p, v) and (f, g) by (f, p) and (p, g).

Lemma 7. *For any corners u and v we have $\delta(u, v) \leq \delta_{G^Q}(u, v) \leq \delta_G(u, v)$. Moreover, if the embedding of a path P in G^Q intersects Q, then P intersects Q in G^Q.*

Proof. Introducing the new intersections and edges can only reduce distances in the graph, and since graph paths in G^Q embed into obstacle avoiding paths, they can never be too short. Finally, the graph G^Q has no edges whose embeddings cross Q. □

We are now left with a pure graph problem; namely to construct connections from each vertex in G^Q to Q with stretch $(1+1/k)$ compared with the distances in G^Q. For the efficiency of the construction, we need to show that G^Q is not too large.

Lemma 8. *A cone edge e can only intersect a separator path Q in a single point unless the cone edge is itself an edge in Q.*

Proof. First, as a matter of definition, if e coincide with a segment of Q, then e is an edge in Q because all corners touched by Q are considered vertices of Q.

Since Q is a shortest path and e a straight line, both avoiding obstacles, we cannot have Q leaving e and coming back to e. Hence, if Q and e were to intersect in more than one point, it would be in a straight line segment $\overline{pq}$. However, if Q arrives e not in the end of e, and turn to follow e, then p is a corner because the shortest path Q only bends at corners. However, then this point is closer to the start of the cone, contradicting the definition of cone edges. □

Lemma 9. *The graph G^Q has at most $(k+1)n$ vertices and at most $3kn+n-1 = O(kn)$ edges. At most n vertices of G^Q are not in Q.*

Proof. The starting point is at most n vertices, at most $n-1$ edges in Q, and at most kn cone edges. These numbers can grow when one of the kn cone edges intersect Q in a single point, as stated in Lemma 8. The subdivisions may lead to 2 extra edges and one extra vertex in Q. □

To get construct the connections, we use the lemma below which is implicit in the proof of [16, Lemma 3.18].

Lemma 10 (Thorup). *Given an arbitrary weighted undirected graph G^Q with a shortest path Q, for each vertex u, we can construct a set $C(u,Q)$ of connections to Q with stretch $1+1/k$ and size $O(1/k)$. The total construction is done via single source shortest path computations in subgraphs $H^{Q'}$ of G^Q except that the shortest path Q' may bypass some vertices in Q. The source is located in Q'. Each vertex appears in $O(k \log n)$ of these subgraphs $H^{Q'}$.*

Using linear time undirected single source shortest paths [4,5], we get

Lemma 11. *For the particular G^Q in our construction, for each vertex u, we can construct a set $C(u,Q)$ of connections to Q with stretch $1+1/k$ and size $O(1/k)$ in $O(nk^2 \log n)$ total time.*

Constructing G^Q. We will now show how to construct G^Q. First we apply the preprocessing of Lemma 3 to the separator path Q alone so that we from any point a in any obstacle boundary direction $360°(i+1/2)/k$ can find the first segment or vertex of Q hit in this direction in $O(\log n)$ time. This preprocessing takes $O(k|Q| \log n)$ time and $O(k|Q|)$ space. For each cone edge (u,v) in G we will now check if and where it intersects Q.

Lemma 12. *If the cone edge (u,v) in cone i is crossed by Q, then it is by a segment (f,g) which is the first part of Q hit from u in one of the two cone boundary directions $360°(i-1/2)/k$ and $360°(i+1/2)/k$.*

Proof. Assume that (u,v) is crossed by Q in cone i from u. We want to prove that the intersection is in one of the segments $(f,g) \in Q$ considered. We will apply Lemma 2 with (u,v) as (a,b). By definition of (u,v), we know that in the original plane with all its obstacles O, the vertex v is as close to u as any visible obstacle corner in cone i from u. If we add Q as an obstacle path, this can only decrease visibility, so v is still as close to u as any visible obstacle corner from $O \cup Q$ in cone i. This uses that all corners of Q were original obstacle corners.

From Lemma 2 we now know that if v is blocked from u by $O \cup Q$, then this is by a segment (f',g') that is also the first segment from $O \cup Q$ hit by one of the cone boundary lines from u in directions $360°(i-1/2)/k$ and $360°(i+1/2)/k$. However, if such an (f',g') blocks v from u, then (f',g') must be from Q since O did not block v from u. Then (f',g') must be the first segment from Q hit by one of the cone boundary lines from u in directions $360°(i-1/2)/k$ and $360°(i+1/2)/k$. Thus (f',g') is one of the segments of Q considered. □

The test in Lemma 12 takes $O(\log n)$ time given the preprocessing from Lemma 2, and from Lemma 8, we know that we only have to find a single intersection point in (u,v). We therefore conclude:

Lemma 13. *We can construct G^Q from G and Q in time $O(kn \log n)$.*

Dealing with all separator paths and recursing. We are now done with all connections to Q, and hence we can delete all edges incident to Q from G^Q including Q itself. More formally, from (1) and Lemma 7 we know that it suffices to estimate distances in G^Q as they are no worse than those in G. Moreover, for any vertices in u and v, $\delta_{G^Q}(u,v) \leq \delta_{G^Q}(u,Q,v) + \delta_{G^Q \setminus V(Q)}(u,v)$. The connections to Q give us an estimate $d(u,Q,v)$ of $\delta_{G^Q}(u,Q,v)$ with stretch $1+1/k$, so now it only remains to estimate distances in $G^Q \setminus V(Q)$. Let G' denote $G^Q \setminus V(Q)$. It is important to note that G' is a subgraph of G. Recursively, this will mean that the edges in G' are the original cone edges whose embedding do not touch any previous separator path.

Next we make connections to the next separator Q', remove it, and continue this way with the up to 6 separator paths in S. When done, we have a graph G^* which is the subgraph of G where we have removed all parts of G whose embedding intersects S. The separator S splits the plane into regions, each with at most half the triangles, and each component of G^* is embedded into one of these regions. We can find out which region in $O(\log n)$ time using the recursion tree $\mathcal{R}$. As in Section 3, we recurse on the regions of the plane obtained by cutting along the separator paths. It might have seemed more natural to recurse separately on each component of G^*, but we measure complexity in terms of the triangles in a region, and if we recursed separately on different components in the same region, the we would have multiple subproblems on the same level sharing the same triangles.

The recursion tree and triangulation was all done in $O(n \log n)$ time. We have $O(n)$ triangles to start with and each triangle occurs in only one subproblem on each level, so the total construction time for a level is $O(kn \log n)$ for constructing the graphs G^Q with Lemma 13 and $O(nk^2 \log n)$ for constructing the connections to separators Q using Lemma 11. Summing over all levels, we get

Lemma 14. *The above construction is done in $O(nk^2(\log n)^2)$ time and $O(kn \log n)$ space.*

Given two corners u and v, the query algorithm is the same as in Section 3, considering all the $O(\log n)$ separator paths connected to u and v, returning the best estimate $d(u, Q, v)$ from Lemma 11.

Lemma 15. *The above query algorithm provides stretch $1 + O(1/k)$ estimates of the original obstacle avoiding distances between corners in $O(k \log n)$ time.*

Proof. The total query time is $O(\log n)$ multiplied with the query time from Lemma 11. For the stretch, consider a shortest u-v path P in the original cone graph G. Then $|P| \leq (1+O(1/k))\delta(u, v)$, The path P remain intact in subgraphs H until the first time we process a separator path Q intersecting the embedding of P. Then the graph H^Q has a path P' with the same embedding as P and which intersects Q, and then the connections from u and v to Q provide an stretch $1 + 1/k$ estimate $d(u, Q, v)$ of $\delta_{H^Q}(u, Q, v) \leq |P|$. Hence $d(u, v) \leq d(u, Q, v) \leq (1 + 1/k)(1 + O(1/k))\delta(u, v) = (1 + O(1/k))\delta(u, v)$. Since any estimate we make corresponds to the length of an obstacle avoiding path in the plane, we know that $d(u, v) \geq \delta(u, v)$. □

Thus we have proved

Proposition 1. *Given a plane with polygonal obstacles, in $O(nk^2(\log n)^2)$ time we can construct an $O(kn(\log n))$ space distance oracle that can provide stretch $(1 + O(1/k))$ distances between obstacle corners in $O(k \log n)$ time.*

We can improve the query time of the above corner oracle to $O(k)$ using the same modifications as in [16] for planar graphs. In [16], the construction time is increased by a factor $k \log n$, but in our case, it only increases by a factor k. The reason is that the modified construction internally use $O(nk \log n)$ connections as auxiliary edges. This is a factor $k \log n$ more edges than in the planar graphs considered in [16], but only a factor $\log n$ more than the $O(kn)$ edges in the cone graphs considered here.

Proposition 2. *Given a plane with polygonal obstacles, in $O(nk^2(\log n)^3)$ time we can construct an $O(kn(\log n))$ space distance oracle that can provide stretch $(1 + O(1/k))$ distances between obstacle corners in $O(k)$ time.*

Plugging this into Chen's construction from Lemma 4, we get answers to arbitrary two point queries in $O(k^3 + k(\log n)/(\log \log n))$ time and with stretch $1+O(1/k)$. To get a desired stretch of $1+\varepsilon$, we choose a large enough $k = O(1/\varepsilon)$. As in [16], paying an extra factor $\log n$ in space,we can also produce a short path in constant time per hop, and even perform routing. This settles Theorem 1.

References

1. Hershberger, J., Suri, S.: An optimal algorithm for Euclidean shortest paths in the plane. SIAM Journal on Computing 28(6), 2215–2256 (1999)
2. Mitchell, J.: Geometric shortest paths and network optimization. In: Handbook of Computational Geometry, pp. 633–701. North Holland, Amsterdam (2000)
3. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice-Hall, Englewood Cliffs (1988)
4. Thorup, M.: Undirected single-source shortest paths with positive integer weights in linear time. J. ACM 46, 362–394 (1999)
5. Thorup, M.: Floats, integers, and single source shortest paths. J. Algorithms 35, 189–201 (2000)
6. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comp. 13(2), 338–355 (1984)
7. Clarkson, K.: Approximation algorithms for shortest path motion planning. In: Proc. 19th STOC, pp. 56–65 (1987)
8. Chen, D.Z.: On the all-pairs Euclidian shortest path problem. In: Proc. 6th SODA, pp. 292–301 (1995)
9. Arikati, S., Chen, D.Z., Chew, L.P., Das, G., Smid, M., Zaroliagis, C.D.: Planar spanners and approximate shortest path queries among obstacles in the plane. In: Díaz, J. (ed.) ESA 1996. LNCS, vol. 1136, pp. 514–528. Springer, Heidelberg (1996)
10. Chiang, Y.J., Mitchell, J.S.B.: Two-point euclidean shortest path queries in the plane. In: Proc. 10th SODA, pp. 215–224 (1999)
11. Gudmundson, J., Levcopoulos, C., Narasimhan, G., Smid, M.: Approximate distance oracles for geometric graphs. In: Proc. 13th SODA, pp. 828–837 (2002)
12. Gudmundson, J., Levcopoulos, C., Narasimhan, G., Smid, M.: Approximate distance oracles revisited. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 357–368. Springer, Heidelberg (2002)
13. Varadajan, K.R.: Personal Communication (2006)
14. Zeh, N.: I/O-Efficient Algorithms for Shortest Path Related Problems. PhD thesis, Carleton University (2002)
15. Klein, P.: Preprocessing an undirected planar network to enable fast approximate distance queries. In: Proc. 13th SODA, pp. 820–827 (2002)
16. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. J. ACM 51(6), 993–1024 (2004) (Announced at FOCS'01)
17. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM J. Appl. Math. 36, 177–189 (1979)
18. Chan, T.M.: Point location in $o(\log n)$ time, Voronoi diagrams in $o(n \log n)$ time, and other transdichotomous results in computational geometry. In: Proc. 47th FOCS, pp. 325–332 (2006)
19. Patrascu, M.: Planar point location in sublogarithmic time. In: Proc. 47th FOCS, pp. 325–332 (2006)

Convex Combinations of Single Source Unsplittable Flows*

Maren Martens[1,**], Fernanda Salazar[2,**], and Martin Skutella[3]

[1] University of British Columbia, Sauder School of Business,
2053 Main Mall, Vancouver, BC V6T1Z2, Canada
maren.martens@sauder.ubc.ca
[2] Escuela Politécnica Nacional, Departamento de Matemática,
Ladrón de Guevara E11-253, Quito, Ecuador
msalazar@math.epn.edu.ec
[3] Universität Dortmund, Fachbereich Mathematik,
44221 Dortmund, Germany
martin.skutella@uni-dortmund.de

Abstract. In the single source unsplittable flow problem, commodities must be routed simultaneously from a common source vertex to certain destination vertices in a given digraph. The demand of each commodity must be routed along a single path. In a groundbreaking paper Dinitz, Garg, and Goemans [4] prove that any given (splittable) flow satisfying certain demands can be turned into an unsplittable flow with the following nice property: In the unsplittable flow, the flow value on any arc exceeds the flow value on that arc in the given flow by no more than the maximum demand.

Goemans conjectures that this result even holds in the more general context with arbitrary costs on the arcs when it is required that the cost of the unsplittable flow must not exceed the cost of the given (splittable) flow. The following is an equivalent formulation of Goemans' conjecture: Any (splittable) flow can be written as a convex combination of unsplittable flows such that the unsplittable flows have the nice property mentioned above. We prove a slightly weaker version of this conjecture where each individual unsplittable flow occurring in the convex combination does not necessarily fulfill the original demands but rounded demands. Preliminary computational results based on our underlying algorithm support the strong version of the conjecture.

1 Introduction

Problem Definition and Notation. The single source unsplittable flow problem was introduced by Kleinberg [8]. We are given a digraph $D = (V, A)$, a source $s \in V$ and K sinks $t_1, \ldots, t_K \in V$. The source and sink nodes are also called *terminals*. We assume without loss of generality that the terminals are pairwise distinct. An unsplittable flow f consists of s-t_i-paths P_i, for $i = 1, \ldots, K$, together with corresponding

* This work was supported in part by the Graduate School of Production Engineering and Logistics, North Rhine-Westphalia, by the DFG Focus Program 1126, Algorithmic Aspects of Large and Complex Networks, grants SK 58/4-1 and SK 58/5-3, and by the German Academic Exchange Service (DAAD).

** Part of this work was done while the authors were at Universität Dortmund.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 395–406, 2007.

flow values $f_i \geq 0$. The flow on arc $a \in A$ is then given as $f(a) = \sum_{i:a \in P_i} f_i$. An unsplittable flow is said to satisfy demands d_i, $i = 1, \ldots, K$, if $f_i = d_i$ for all i.

We also consider flows from s to the sinks $t_1, \ldots, t_K$ that are not necessarily unsplittable. Such a flow f is given by flow values $f(a)$ for all arcs $a \in A$ such that flow conservation constraints for all non-terminal nodes are met. Moreover, the net amount of flow leaving the source s as well as the net amount of flow arriving at each sink t_i must be non-negative. In order to emphasize that a flow is not unsplittable we sometimes also call it a *splittable flow*. A splittable flow satisfies demands d_i, $i = 1, \ldots, K$, if the net amount of flow arriving at t_i equals d_i for all i. The flow traveling from the source to sink t_i is sometimes also referred to as *commodity* i.

Dinitz, Garg, and Goemans [4] present an algorithm that turns a given splittable flow f^{init} satisfying demands d_i, $i = 1, \ldots, K$, into an unsplittable flow f satisfying the same demands such that

$$f(a) \;\leq\; f^{\text{init}}(a) + d_{\max} \qquad \text{for all arcs } a \in A \tag{1}$$

where $d_{\max} := \max_{i=1,\ldots,K} d_i$. Goemans conjectures that this result can be generalized as follows.

Conjecture 1 (Goemans [6]). For any cost function $c : A \to \mathbb{R}$, a splittable flow f^{init} satisfying given demands d_i, $i = 1, \ldots, K$, can be turned into an unsplittable flow f satisfying the same demands such that property (1) holds and the cost of f is bounded by the cost of f^{init}, i.e.,

$$\sum_{a \in A} c(a) f(a) \;\leq\; \sum_{a \in A} c(a) f^{\text{init}}(a).$$

Network flows are usually considered in digraphs with arc capacities $u : A \to \mathbb{R}_0^+$. The capacity $u(a)$ of arc a is an upper bound on the amount of flow that can be sent through arc a. Moreover, for unsplittable flow problems it is often assumed that all demands are at most as large as the minimum arc capacity, i.e., $d_{\max} \leq u_{\min} := \min_{a \in A} u(a)$ such that any commodity can in principle be routed through any arc unsplittably. This condition is also known as the *balance condition*. If the balance condition is fulfilled and f^{init} obeys given arc capacities, then it follows from property (1) that the unsplittable flow f has congestion at most 2, i.e., $f(a) \leq 2u(a)$ for all arcs $a \in A$. In particular, the algorithm presented by Dinitz et al. [4] achieves performance ratio 2 for the objective to minimize congestion.

Related Results from the Literature. The single source unsplittable flow problem is a special case of the more general unsplittable flow problem (UFP) where each commodity has its own source and sink. This problem has been well studied in the literature. In the case that we are given arc capacities and demands for each commodity and look for an unsplittable flow of minimum congestion, i.e., of minimum overload of arc capacities, Raghavan and Thompson [13,12] introduce a randomized rounding technique which yields an $O(\log m / \log \log m)$-approximation algorithm provided that the balance condition[1] holds. Here, m is the number of arcs in the underlying graph. Chuzhoy

[1] Unless stated otherwise, the balance condition is always assumed to be met for the UFP.

and Naor [3] show that the directed case of the UFP is $\Omega(\log\log m)$-hard to approximate unless $NP \subseteq DTIME(n^{O(\log\log\log n)})$, where n is the number of vertices in the underlying graph. Before this result was found, only APX-hardness for the UFP was known (see, e.g., Kleinberg [8]). In the special case of unit demands and unit edge capacities (the *edge-disjoint paths problem*) Andrews and Zhang [1] prove that there is no $(\log\log m)^{1-\epsilon}$-approximation for the undirected congestion minimization problem, unless $NP \subseteq ZPTIME(n^{polylog\ n})$.

For the optimization problem to route a subset of commodities whose total sum of demands is maximal, Azar and Regev [2] present a strongly polynomial algorithm with approximation ratio $O(\sqrt{m})$. Kolman and Scheideler [10] even give a strongly polynomial $O(\sqrt{m})$-approximation algorithm for the problem without the balance condition. On the other hand, Guruswami, Khanna, Rajaraman, Shepherd, and Yannakakis [7] show that there is no approximation algorithm with performance ratio $O(m^{\frac{1}{2}-\epsilon})$ for any $\epsilon > 0$, unless $P = NP$.

It is an easy observation that already the single source unsplittable flow problem without costs contains several well-known NP-complete problems as special cases, such as, for example, PARTITION, BIN PACKING, or even scheduling parallel machines with makespan objective [11]. If we consider the problem with costs, we obtain the KNAPSACK problem as a special case. We refer to [4,8,9,14] for more details and other special cases.

Kleinberg [8], Dinitz, Garg, and Goemans [4], Kolliopoulos and Stein [9], and Skutella [14] present approximation algorithms for various optimization versions of the single source unsplittable flow problem. Du and Kolliopoulos [5] have implemented and empirically tested several of those approximation algorithms. As already mentioned above, a 2-approximation algorithm for congestion minimization is given in [4]. In [14], a 3-approximation algorithm is presented for the corresponding problem with costs. It is also shown there that the performance ratio can be decreased to 2 if the demands are multiples of each other, i.e., $d_i | d_j$ or $d_j | d_i$ for all $i, j = 1, \ldots, K$. In fact, it is shown in [14] that Conjecture 1 holds in this special case. For arbitrary demands, a weaker version of Conjecture 1 is shown where property (1) is replaced with the following less restrictive property: $f(a) \leq 2f^{\text{init}}(a) + d_{\max}$ for all arcs $a \in A$.

Contribution of this Paper. It is not difficult to observe that the following conjecture is equivalent to Conjecture 1.

Conjecture 2. A splittable flow f^{init} satisfying given demands d_i, $i = 1, \ldots, K$, can be written as a convex combination of unsplittable flows satisfying the same demands such that property (1) holds for each unsplittable flow f occurring in the convex combination.

We argue briefly that the two conjectures are indeed equivalent. It is easy to see that Conjecture 1 holds if Conjecture 2 is true. On the other hand, if Conjecture 2 is false, then there exists a splittable flow f^{init} that is not contained in the convex hull of the set of unsplittable flows f satisfying property (1) and the same demands as f^{init}. In this case there must exist a separating hyperplane whose normal vector yields a cost function c such that the cost of f^{init} with respect to c is strictly smaller than the cost of every unsplittable flow under consideration. This contradicts Conjecture 1.

Unfortunately we are not able to prove Goemans' Conjecture in this paper but we show the following slightly weaker version.

Theorem 1. *A splittable flow f^{init} satisfying given demands d_i, $i = 1, \ldots, K$, can be written as a convex combination of unsplittable flows such that property (1) holds for each unsplittable flow f occurring in the convex combination.*

The only difference to Conjecture 2 is that we cannot guarantee that the unsplittable flows occurring in the convex combination satisfy the same demands as the given flow f^{init}. Instead the demands satisfied by an individual unsplittable flow are the original demands rounded (up or down) by a factor of at most 2 to the next $d_{max}/2^{\ell}$ with $\ell \in \mathbb{N}$. Here, d_{max} is the maximum original demand.

In Section 2 we present an algorithm that, given a splittable flow f^{init}, computes unsplittable flows together with appropriate weights such that the resulting convex combination (weighted sum) is equal to the given flow f^{init}. The algorithm uses ideas of Kolliopoulos and Stein [9] and Skutella [14]. It builds a binary tree whose root node is the given splittable flow f^{init} and whose leafs are unsplittable flows fulfilling property (1). Moreover, each non-leaf node is a convex combination of its two children. As a consequence, the splittable flow f^{init} at the root is a convex combination of the unsplittable flows at the leafs.

We give a detailed analysis of the algorithm proving its correctness and the existence of the requested unsplittable flows in Section 3. Finally, in Section 4 we discuss preliminary computational results trying to confirm that there also exist convex combinations of unsplittable flows of additive congestion at most d_{max} if we restrict to using the original demands instead of the rounded ones that are produced by our algorithm. We also observe in this section that a convex combination as described in Theorem 1 can be computed in polynomial time.

2 Constructing the Convex Combination

We present an algorithm that, given a splittable flow f^{init}, computes unsplittable flows with weights such that the weighted sum (convex combination) of the unsplittable flows equals f^{init}. These unsplittable flows have property (1) and they satisfy demands that are obtained by rounding the ones in f^{init}.

2.1 Preliminaries

To simplify the description of the algorithm we introduce some more vocabulary. We say that a *terminal path* is a maximal (not necessarily directed) simple path in D with endpoints in $\{s, t_1, \ldots, t_K\}$, where *maximal* means that it is not extendable at either of its endpoints. A (not necessarily directed) cycle or a terminal path is called an *augmenting structure*.

We use $d_{max} := \max_{i=1,\ldots,K} d_i$ ($d_{min} := \min_{i=1,\ldots,K} d_i$) to denote the maximum (minimum) demand satisfied by f^{init}. For a natural number ℓ, we define $r_f^{\ell}(a) := f(a) \mod \frac{d_{max}}{2^{\ell}}$, for $a \in A$. A flow is called *q-integral* for some $q \in \mathbb{R}^+$, if the flow value on each arc is an integral multiple of q. It is well known that any single source multicommodity flow that is q-integral can be decomposed into flows on paths with flow value q.

The definition $f(a) := \sum_{i:a \in P_i} f_i$, for all $a \in A$, enables us to build the sum $f+g$ of two flows f and g arcwise, even if one or both flows are unsplittable and given pathwise.

2.2 The Algorithm

In the following we shortly sketch the idea of the algorithm first and give a more detailed characterization of it later on: We want to round each demand to the two nearest (upper and lower) values of the form $d_{max}/2^{\ell}$, for some $\ell \in \mathbb{N}$, and then send these rounded demands unsplittably. A convex combination of the final unsplittable flows (with rounded demands) will yield our original flow. The largest value we need to consider for ℓ is

$$L := \left\lceil \log \frac{d_{max}}{d_{min}} \right\rceil,$$

since $d_{max}/2^L$ is the largest fraction $d_{max}/2^{\ell}$ ($\ell \in \mathbb{N}$) that is at most d_{min}. (Thus, $d_{max}/2^L$ is the smallest demand obtained from rounding all demands to $(d_{max}/2^{\ell})$-integrality, for some $\ell \in \mathbb{N}$).

We start by making the input flow $(d_{max}/2^L)$-integral. This is done by augmenting flow on cycles and terminal paths. The particular augmenting structure and the increment of flow are chosen such that after augmentation the flow on at least one additional arc is $(d_{max}/2^L)$-integral. Augmentation on a cycle does not change the demands, whereas augmentations on terminal paths yield changes of demands to $(d_{max}/2^L)$-integrality. We always augment in both directions of an augmenting structure and thus obtain two new flows in each step such that the "parent" flow is a convex combination of the two. Depending on the direction in which flow is augmented on a path, demands are rounded up or down. (See Figure 1 for an illustration of the algorithm.) Considering the two direct descendants of a "parent" flow, which result from a change of flow in either direction of an augmenting structure, we simply construct a binary tree of flows in which the descendants of a flow f can be used to form a convex combination of f. The leaves of this tree finally form the desired convex combination of unsplittable flows for the root flow f^{init}.

When a flow is $(d_{max}/2^L)$-integral, demands of value $d_{max}/2^L$ can be satisfied unsplittably. For this reason, we want to prevent such demands and corresponding flow carrying paths from further changes. Thus, for each sink t_i with demand $d_{max}/2^L$, we decrease the current flow along a corresponding flow carrying s-t_i-path by $d_{max}/2^L$. These paths with the flow values are stored for all subsequent flows. After the decrement, all demands are at least $d_{max}/2^{L-1}$ and we turn to make the (remaining) flow $(d_{max}/2^{L-1})$-integral. We proceed in this manner until the (remaining) flow is d_{max}-integral. Then all (remaining) demands equal d_{max} and are served unsplittably.

For the sake of simple presentation and analysis of the algorithm, we give a recursive description of it in Algorithm 1. The initial call to start the recursive algorithm is given by DECOMP(D,f^{init},$\emptyset$,1) where f^{init} is the single source multicommodity flow in the digraph D that we want to write as a convex combination of unsplittable flows. The third parameter is a set of paths with corresponding flow values. If (P, ϕ) is in this set, it means that all subsequent flows route ϕ units of flow along path P. The last parameter indicates the weight of the flow that is to be decomposed. For the initial flow this weight equals 1.

A call of DECOMP(D,f^{init},$\emptyset$,1) effects the following: In each step of the recursion, it first updates the input flow f by iteratively deleting demands $d_{max}/2^{\ell}$, for $\ell \in \mathbb{N}$, if f is $(d_{max}/2^{\ell})$-integral. The related flow carrying paths are added to the current

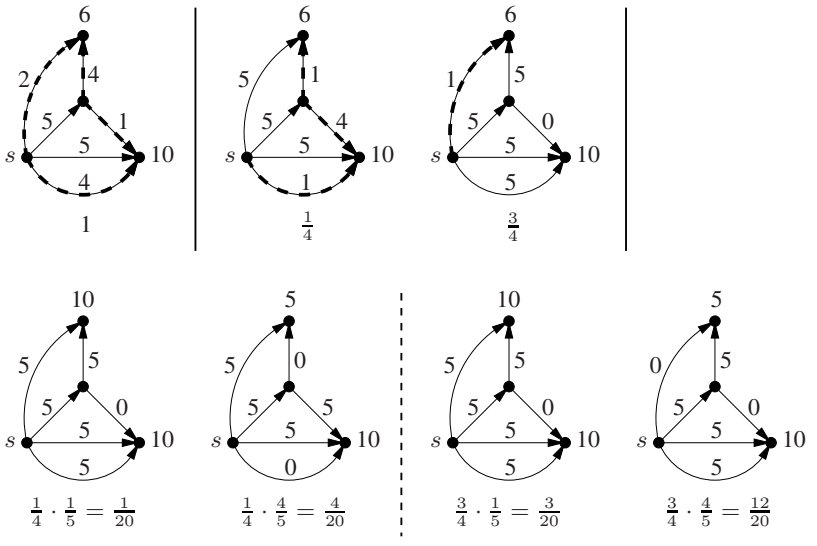


Fig. 1. Steps of the algorithm to 5-integrality. We start with the flow in the upper left corner and read from left to right first. Since $d_{max}/2^L$ equals 5, we need to make the flow 5-integral. The dashed arcs indicate the augmenting structure that is used. Since we augment flow in either direction of an augmenting structure, each non-5-integral flow produces two new flows that are separated from their "parents" by a vertical line. We consider the first flow. If we use the indicated augmenting structure clockwise, we may augment by 3. Then the arc with flow value 2 becomes 5-integral. Using the augmenting structure counterclockwise, we may augment by 1. Then the unit of flow on the arc with flow value 1 recedes. The number below each flow indicates its weight in the convex combination.

unsplittable flow given by $\mathcal{P}$. Afterwards it decomposes the (remaining) flow f into a convex combination of two new flows f_1 and f_2 that result from a single augmentation in both directions of a suitable augmenting structure. Further steps of the recursion decompose f_1 and f_2 into convex combinations of unsplittable flows.

The augmentation itself works as follows. Let us assume that the flow f that remains after the update is $(d_{max}/2^\ell)$- but not $(d_{max}/2^{\ell-1})$-integral, for some $\ell \in \{1, \ldots, L\}$. Then we consider the subgraph $\tilde{D}$ of D that consists of all arcs $a \in A$ whose flow values $f(a)$ are not $(d_{max}/2^{\ell-1})$-integral. Starting from an arbitrary arc we follow an undirected path (in either direction) until there is no more incident arc or we get to a node which has already been visited. The first criterion results in a terminal path, the second one in a cycle. (We prove this later on in Lemma 2.) For the resulting augmenting structure C we augment by the minimum δ of gaps between flow values and the next lower multiples of $d_{max}/2^{\ell-1}$ for arcs that are used by C in backward direction and of gaps between flow values and the next upper multiples of $d_{max}/2^{\ell-1}$ for arcs that are used by C in forward direction. Defining C^- as the backward arcs in C and C^+ as the forward arcs in C we can write δ as follows.

$$\delta = \min\{\min_{a \in C^-} r_f^{\ell-1}(a), \min_{a \in C^+} \frac{d_{max}}{2^{\ell-1}} - r_f^{\ell-1}(a)\}$$

Algorithm 1. DECOMP(D,f,$\mathcal{P}$,w)

Input: A digraph $D = (V, A)$, a single source multicommodity flow f in D with source s, a set $\mathcal{P}$ of paths from s to pairwise distinct nodes t in D with corresponding flow values, and a positive weight $w \in (0, 1]$. The maximum (minimum) demand, that f satisfies, is d_{max} (d_{min}).

Output: A set of unsplittable flows with weights that sum up to w yielding a conic combination of $w(f + f_{\mathcal{P}})$, where $f_{\mathcal{P}}$ denotes the flow that is given by $\mathcal{P}$.

for $i = \left\lceil \log \frac{d_{max}}{d_{min}} \right\rceil$ **downto** 0 **do**
  if f *is* $(d_{max}/2^i)$*-integral* **then**
    for *each sink* t *having demand* $d = d_{max}/2^i$ *in* f **do**
      Determine an arbitrary flow carrying s-t-path P in D.
      Set $f(a) := f(a) - d$ for all $a \in P$.
      Set $\mathcal{P}' := \mathcal{P} \cup \{(P, d)\}$.
    end
  end
end
if $f \equiv 0$ **then**
  return $(\mathcal{P}', w)$.
end
Set $\ell := \min\{\min\{j \in \mathbb{N} | f \text{ is } (d_{max}/2^j)\text{-integral}\}, \left\lceil \log \frac{d_{max}}{d_{min}} \right\rceil + 1\}$.
Let $C \subseteq A$ be an augmenting structure with $r_f^{\ell-1}(a) \neq 0 \; \forall \, a \in C$.
Set $C^+ := \{a \in C \mid C \text{ traverses } a \text{ in forward direction}\}$,
  $C^- := \{a \in C \mid C \text{ traverses } a \text{ in backward direction}\}$.
Set $\delta_1 := \min\{\min_{a \in C^-} r_f^{\ell-1}(a), \min_{a \in C^+} \frac{d_{max}}{2^{\ell-1}} - r_f^{\ell-1}(a)\}$,
  $\delta_2 := \min\{\min_{a \in C^+} r_f^{\ell-1}(a), \min_{a \in C^-} \frac{d_{max}}{2^{\ell-1}} - r_f^{\ell-1}(a)\}$.
For $a \in A \setminus C$ set $f_1(a) := f(a)$ and $f_2(a) := f(a)$.
For $a \in C^+$ set $f_1(a) := f(a) + \delta_1$ and $f_2(a) := f(a) - \delta_2$.
For $a \in C^-$ set $f_1(a) := f(a) - \delta_1$ and $f_2(a) := f(a) + \delta_2$.
Set $w_1 := \frac{\delta_2}{\delta_1+\delta_2} w$ and $w_2 := \frac{\delta_1}{\delta_1+\delta_2} w$.
return DECOMP(D,f_1,$\mathcal{P}'$,w_1) $\cup$ DECOMP(D,f_2,$\mathcal{P}'$,w_2).

Now let f_1 and f_2 be the flows resulting from augmenting along C and its "counterpart", i.e., C in the opposite direction. Let δ_1 and δ_2 be the corresponding augmentation values. Then the weight of f_i (for $i = 1, 2$) is given by the weight of f multiplied with $\delta_{3-i}/(\delta_1 + \delta_2)$.

3 Analysis of the Algorithm

In Section 3.1 we show that the flows and weights returned by DECOMP(D,f^{init},$\emptyset$,1) yield a convex combination for f^{init}. Further, the produced flows are unsplittable and all its demands are of the form $d_{max}/2^\ell$, for some $\ell \in \{0, \ldots, \lceil \log(d_{max}/d_{min}) \rceil\}$. To prove Theorem 1 we have to show as well that, on each arc $a \in A$, the final flows send at most the initial flow $f^{\text{init}}(a)$ plus an additive d_{max}. This is done in Section 3.2.

3.1 Correctness of the Algorithm

It is easy to see that the following lemma is true. A detailed proof is omitted due to space limitations.

Lemma 1. *For each DECOMP(D,f,$\mathcal{P}$,w) that is triggered by DECOMP(D,f^{init},$\emptyset$,1) it holds that*

1. *f is a single source multicommodity flow in D,*
2. *$\mathcal{P}$ is a set of paths from the source of f to pairwise distinct nodes t in D with corresponding flow values, and*
3. *$w \in (0, 1]$.*

Lemma 1 shows that the algorithm is well-defined. The following lemma is necessary to prove that our algorithm terminates. It follows immediately from flow conservation.

Lemma 2. *If a flow f in D is not $(d_{max}/2^{\ell})$-integral, for some $\ell \in \mathbb{N}$, then there exists an augmenting structure $C \subseteq A$ with $r_f^{\ell}(a) \neq 0$, for all $a \in C$.*

The definition of δ_1 and δ_2 and the augmentation rule imply that if f is not decreased in the for-loop of DECOMP(D,f,$\mathcal{P}$,w), the flows f_1 and f_2 are "more integral" than f.

Lemma 3. *For any flow f that is augmented in Algorithm 1 and its corresponding value ℓ as defined in the algorithm, it holds that f_1 and f_2 each have at least one more arc than f whose flow value is $(d_{max}/2^{\ell-1})$-integral.*

Before we turn to proving that the flows/weights returned by DECOMP(D,f,$\mathcal{P}$,w) yield a conic combination of wf, we show that the procedure indeed terminates and outputs unsplittable flows whose demands are of the form $d_{max}/2^{\ell}$, for some $\ell \in \{0, \ldots, \lceil \log(d_{max}/d_{min}) \rceil\}$.

Corollary 1. *DECOMP(D,f,$\mathcal{P}$,w) terminates. The output is a set of unsplittable flows whose demands are of the form $d_{max}/2^{\ell}$, for some $\ell \in \{0, \ldots, \lceil \log(d_{max}/d_{min}) \rceil\}$.*

Proof. We proved that the flows f_1 and f_2 resulting from DECOMP(D,f,$\mathcal{P}$,w) have fewer positive demands than f or are "more integral". The first property eventually results in a decrement of the input flow to the zero flow. In every recursive call of Algorithm 1 in which the number of positive demands is not decreased for the respective input, the flow value on at least one of its arcs changes to a "higher" integrality. After at most $|A|$ steps we therefore change from $(d_{max}/2^{\ell})$-integrality to $(d_{max}/2^{\ell-1})$-integrality for some $\ell \in \{1, \ldots, L\}$ (or respectively from the initial state to $(d_{max}/2^{L})$-integrality). At this point demands of value $(d_{max}/2^{\ell-1})$ and corresponding flow are deleted in the for-loop. If no such demands exist, we go to "higher" integralities and delete demands at the latest when the flow is d_{max}-integral. Therefore, at some point all demands are deleted and the algorithm terminates.

It follows from the preceding analysis that all paths in the final $\mathcal{P}'$ connect the source in f with pairwise distinct sinks. Thus, $\mathcal{P}'$ yields an unsplittable flow. It follows directly from the specification of the algorithm that all demands served by $\mathcal{P}'$ are of the form $d_{max}/2^{\ell}$, for some $\ell \in \{0, \ldots, \lceil \log(d_{max}/d_{min}) \rceil\}$.

In the following we use $f_{\mathcal{P}}$ to denote the flow that is given by some set $\mathcal{P}$ of paths with corresponding flow values. We prove the following helpful lemma in order to show that DECOMP(D,f,$\mathcal{P}$,w) returns the specified output.

Lemma 4. *Consider DECOMP(D,f,$\mathcal{P}$,w). It holds that*

$$w(f + f_{\mathcal{P}}) = w_1(f_1 + f_{\mathcal{P}'}) + w_2(f_2 + f_{\mathcal{P}'}) \tag{2}$$

and $w_1 + w_2 = w$.

Proof. The second part of the lemma follows immediately from the definition of w_1 and w_2. Equation (2) can be proven as follows. For all arcs $a \in A$ that are not in the augmenting structure C that leads from f to f_1 and f_2, it holds that $f_1(a) = f_2(a) = f(a) - (f_{\mathcal{P}'}(a) - f_{\mathcal{P}}(a))$. Since $w_1 + w_2 = w$, equation (2) follows immediately.

Now consider an arc $a \in C^+$. It holds that $f_1(a) = f(a) - (f_{\mathcal{P}'}(a) - f_{\mathcal{P}}(a)) + \delta_1$ and $f_2(a) = f(a) - (f_{\mathcal{P}'}(a) - f_{\mathcal{P}}(a)) - \delta_2$. Using $w_1 + w_2 = w$, it follows that $w_1 f_1(a) + w_2 f_2(a) = wf(a) + wf_{\mathcal{P}}(a) - w_1 f_{\mathcal{P}'}(a) - w_2 f_{\mathcal{P}'}(a)$. The proof is analogous for $a \in C^-$.

The following corollary demonstrates that the output of DECOMP(D,f,$\mathcal{P}$,w) is correct. With this result we are finished proving the correctness of the algorithm as described in Section 2. To prove our main result we still need to show that all unsplittable flows that are returned by DECOMP(D,f^{init},$\emptyset$,1) have congestion at most 2. This is done in Theorem 2 in Section 3.2.

Corollary 2. *The flows and weights returned by DECOMP(D,f,$\mathcal{P}$,w) yield a conic combination of $w(f + f_{\mathcal{P}})$ whose weights sum up to w.*

The proof of Corollary 2 uses induction on the depth of recursion and Lemma 4. It is omitted due to space limitations. The next corollary follows immediately.

Corollary 3. *The flows and weights returned by DECOMP(D,f^{init},$\emptyset$,1) yield a convex combination of f.*

3.2 Upper Bound on the Congestion

Together with the algorithm from Section 2 and its analysis in Section 3.1 the following theorem is the last component to prove Theorem 1.

Theorem 2. *For a single source multicommodity flow f^{init} in $D = (V, A)$ and any arc $a \in A$, it holds that the flow along a in any flow produced by DECOMP(D,f^{init},$\emptyset$,1) exceeds $f^{init}(a)$ by at most an additive d_{max}.*

Proof. Consider the progression of the input flow while DECOMP(D,f^{init},$\emptyset$,1) is running. Let f^0 be a flow that occurs on the way to $(d_{max}/2^L)$-integrality of the input flow. Further, let $\mathcal{P}^0$ be the current unsplittable flow while f^0 is considered in the algorithm.

By the choice of δ_1 and δ_2, it holds for all $a \in A$ that

$$f^0(a) + f_{\mathcal{P}^0}(a) \leq f^{\text{init}}(a) + \frac{d_{max}}{2^L} - \left(f^{\text{init}}(a) \mod \frac{d_{max}}{2^L} \right), \tag{3}$$

because once the flow on a is $(d_{max}/2^L)$-integral, i.e., rounded to at most the next multiple of $d_{max}/2^L$, it is not changed again on the way to $(d_{max}/2^L)$-integrality.

After $(d_{max}/2^L)$-integrality was reached, the input flow is iteratively augmented to $(d_{max}/2^{L-\ell})$-integrality for gradually increasing $\ell \in \{1, \dots, L\}$. Let f^ℓ be a flow that occurs while DECOMP(D,f^{init},$\emptyset$,1) is running and that is $(d_{max}/2^{L-\ell})$-integral, but not $(d_{max}/2^{L-\ell-1})$-integral. Further, let $f^{\ell-1}$ be any ancestor of f^ℓ, i.e., any of the flows that (indirectly) caused the creation of f^ℓ, that is $(d_{max}/2^{L-\ell+1})$-integral. Again let $\mathcal{P}^\ell$ and $\mathcal{P}^{\ell-1}$ be the corresponding unsplittable flows.

In analogy with (3), it follows from the choice of δ_1 and δ_2 that for all $a \in A$

$$f^\ell(a) + f_{\mathcal{P}^\ell}(a) \leq f^{\ell-1}(a) + f_{\mathcal{P}^{\ell-1}}(a) + \frac{d_{max}}{2^{L-\ell}} - \frac{d_{max}}{2^{L-\ell+1}}.$$

We can prove an analogous equation if some integrality step is omitted, i.e., if there is no ancestor of f^ℓ that is $(d_{max}/2^{L-\ell+1})$-integral. Let ℓ' be the largest integer that is smaller than ℓ and for which an ancestor of f^ℓ exists that is $(d_{max}/2^{L-\ell'})$-integral. Then it holds that $f^\ell(a) + f_{\mathcal{P}^\ell}(a) \leq f^{\ell'}(a) + f_{\mathcal{P}^{\ell'}}(a) + d_{max}/2^{L-\ell} - d_{max}/2^{L-\ell'}$.

We obtain iteratively, for all $\ell \in \{0, \dots, L\}$, that

$$f^\ell(a) + f_{\mathcal{P}^\ell}(a) \leq f^{\text{init}}(a) + \frac{d_{max}}{2^{L-\ell}} - \left(f^{\text{init}}(a) \mod \frac{d_{max}}{2^L} \right). \tag{4}$$

Now consider the point when the flow f is changed to $f' \equiv 0$ in the for-loop. Let $\mathcal{P}$ and $\mathcal{P}'$ be the corresponding unsplittable flows. Then $\mathcal{P}'$ is one of the output flows of the algorithm. Since d_{max} is the maximum demand in f, it follows that f is d_{max}-integral. With (4) we have $f_{\mathcal{P}'}(a) = f(a) + f_{\mathcal{P}}(a) \leq f^{\text{init}}(a) + d_{max} - (f^{\text{init}}(a) \mod \frac{d_{max}}{2^L}) \leq f^{\text{init}}(a) + d_{max}$.

Note that it even holds, for all $a \in A$, that $f_{\mathcal{P}'}(a) < f^{\text{init}}(a) + d_{max}$. To obtain this result, we have to regard that $f^0(a) + f_{\mathcal{P}^0}(a) \leq f^{\text{init}}(a)$, if $f^{\text{init}}(a)$ is $(d_{max}/2^L)$-integral.

If we assume f^{init} to be feasible, the next result follows immediately.

Corollary 4. *If a single source multicommodity flow f^{init} in $D = (V, A)$ obeys arc capacities $u : A \to \mathbb{R}^+$ and the balance condition is met, then all flows produced by DECOMP(D,f^{init},$\emptyset$,1) have congestion at most 2.*

We close this section with an example showing that our result is tight (see also [4] for similar results).

Lemma 5. *There exists a network and a feasible fractional single source multicommodity flow f^{init} such that in each convex combination of unsplittable flows forming f^{init} there is at least one flow with congestion arbitrarily close to 2.*

Proof. Consider a network with source s, sinks t_1, t_2 with demands 1 for both commodities, and one additional node v. The arcs in the network with their initial flow values are (s, v) with $f^{\text{init}}((s, v)) = 1 + \epsilon$, (s, t_2) with $f^{\text{init}}((s, t_2)) = 1 - \epsilon$, (v, t_1) with $f^{\text{init}}((v, t_1)) = 1$, and (v, t_2) with $f^{\text{init}}((v, t_2)) = \epsilon$. The capacity of arc a is the maximum of $f^{\text{init}}(a)$ and 1. ϵ is an arbitrary positive number smaller than 1.

Consider arc (s, v). Obviously we have to route commodity 1 on it in each unsplittable flow that participates in a convex combination forming f^{init}. But there must also be at least one unsplittable flow that routes commodity 2 on this arc. Thus, we obtain a flow value of 2 on it and a congestion of $2/(1+\epsilon)$.

4 Some Preliminary Computational Results

We have shown that it is possible to write any (splittable) single source multicommodity flow f^{init} as a convex combination of unsplittable flows obeying condition (1). The demands satisfied by the unsplittable flows that we construct for this convex combination slightly differ from the ones in the original flow.

Using our algorithm, we want to empirically confirm Conjecture 2. In principle, we would like to do the following: Consider all unsplittable flows computed by the algorithm; turn them into unsplittable flows satisfying the original demands d_i, $i = 1, \ldots, K$, by simply routing exactly d_i units of flow along the chosen s-t_i-paths (instead of the rounded demand values); omit all unsplittable flows which, after this modification, no longer obey condition (1) (notice that this can easily happen already for simple examples); check whether f^{init} is contained in the convex hull of the remaining unsplittable flows.

The main problem with this approach is the huge size of the binary tree computed by our algorithm. Already for relatively small instances the algorithm does not terminate in reasonable time due to the exponential growth of the computed binary tree. It is therefore not realistic to try to compute all unsplittable flows corresponding to leaf nodes of that tree. Instead, we have to thin out the tree and only compute a subset of leafs of reasonable size. Of course, this subset should still have the property that f^{init} is contained in the convex hull of unsplittable flows given by the subset.

More precisely, we proceed as follows. We start to compute the binary tree in breadth-first manner. When we arrive at a layer of the tree containing "too many" nodes, we omit some of them and only maintain a subset of "reasonable size". By construction the flow f^{init} at the root node is a convex combination of the flows corresponding to the nodes of the tree in any fixed layer. It follows from Carathéodory's theorem that f^{init} can be written as a convex combination of a subset containing at most $|A| + 1$ flows. It is therefore possible to keep the width of the tree bounded by $O(|A|)$ and still maintain the property that f^{init} is a convex combination of the flows in any fixed layer. In our implementation, we simply use CPLEX to find suitable subsets of flows when we arrive at a layer of the tree that contains "too many" nodes. Since the depth of the tree is polynomially bounded, we can even find a representation of f^{init} as a convex combination of unsplittable flows in polynomial time and thus strengthen Theorem 1 as follows.

Theorem 3. *A convex combination as described in Theorem 1 can be obtained in polynomial time.*

For the purpose of our empirical study it is not advisable to reduce the width of each layer of the tree as far as possible (i.e., to width $|A| + 1$). This decreases our chance to find sufficiently many "good" unsplittable flows in the end that contain f^{init} in their convex hull. Therefore the challenge is to decide for each layer of the tree which flows to keep and which to omit in order to keep the width of the tree small. We have ex-

perimented with several different strategies but have not found the ideal strategy yet. However, for most instances that we consider our choice of flows is suitable in the sense that the resulting unsplittable flows meet the congestion requirement.

For the empirical tests we use 14 test instances that were also considered by Du and Kolliopoulos [5] for their empirical evaluation of approximation algorithms. The instances come from the following generators: noigen, satgen, rangen, and genrmf. A complete description of every generator can be found in [5]. We have tested different ways to choose the flows that we keep in our convex combinations. So far, for half of the 14 instances we were able to compute an appropriate set of unsplittable flows of additive congestion at most d_{max}. For another three instances the multiplicative congestion of the unsplittable flows does not exceed 2, while for the remaining four instances we have not found an appropriate set of unsplittable flows yet. We are currently still working on finding better heuristics to choose the flows to be kept in each layer of the tree.

References

1. Andrews, M., Zhang, L.: Hardness of the undirected congestion minimization problem. In: Proceedings of 37th Annual ACM Symposium on Theory of Computing, pp. 284–293. ACM Press, New York (2005)
2. Azar, Y., Regev, O.: Strongly polynomial algorithms for the unsplittable flow problem. In: Proceedings of the 8th Conference on Integer Programming and Combinatorial Optimization, pp. 15–29 (2001)
3. Chuzhoy, J., Naor, J.: New hardness results for congestion minimization and machine scheduling. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, pp. 28–34. ACM Press, New York (2004)
4. Dinitz, Y., Garg, N., Goemans, M.X.: On the single source unsplittable flow problem. Combinatorica 19, 17–41 (1999)
5. Du, J., Kolliopoulos, S.: Implementing approximation algorithms for the single-source unsplittable flow problem. Journal of Experimental Algorithmics 10, 2–3 (2005)
6. Goemans, M.X.: Cited as personal communication from (January 2000) [14, Section 7]
7. Guruswami, V., Khanna, S., Rajaraman, R., Shepherd, B., Yannakakis, M.: Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In: Proceedings of the 31st Annual ACM Symposium on Theory of Computing, pp. 19–28. ACM Press, New York (1999)
8. Kleinberg, J.M.: Approximation Algorithms for Disjoint Path Problems. PhD thesis, Massachusetts Institute of Technology (May 1996)
9. Kolliopoulos, S.G., Stein, C.: Approximation algorithms for single-source unsplittable flow. SIAM Journal on Computing 31, 919–946 (2002)
10. Kolman, P., Scheideler, C.: Improved bounds for the unsplittable flow problem. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 184–193. ACM Press, New York (2002)
11. Lenstra, J.K., Shmoys, D.B., Tardos, É.: Approximation algorithms for scheduling unrelated parallel machines. Mathematical Programming 46, 259–271 (1990)
12. Raghavan, P.: Probabilistic construction of deterministic algorithms: approximating packing integer programs. Journal of Computer and System Sciences 37, 130–143 (1988)
13. Raghavan, P., Thompson, C.D.: Randomized rounding: A technique for provably good algorithms and algorithmic proofs. Combinatorica 7, 365–374 (1987)
14. Skutella, M.: Approximating the single source unsplittable min-cost flow problem. Mathematical Programming 91, 493–514 (2002)

Farthest-Polygon Voronoi Diagrams*

Otfried Cheong[1], Hazel Everett[2], Marc Glisse[2], Joachim Gudmundsson[3], Samuel Hornus[1], Sylvain Lazard[2], Mira Lee[1], and Hyeon-Suk Na[4]

[1] Dept. of Computer Science, KAIST, Daejeon, Korea
{otfried,hornus,mira}@tclab.kaist.ac.kr
[2] LORIA – INRIA Lorraine, Université Nancy 2, Nancy, France
Firstname.Name@loria.fr
[3] National ICT Australia Ltd.**, Sydney, Australia
joachim.gudmundsson@nicta.com.au
[4] School of Computing, Soongsil University, Seoul, Korea
hsnaa@ssu.ac.kr

Abstract. Given a family of k disjoint connected polygonal sites of total complexity n, we consider the farthest-site Voronoi diagram of these sites, where the distance to a site is the distance to a closest point on it. We show that the complexity of this diagram is $O(n)$, and give an $O(n \log^3 n)$ time algorithm to compute it.

1 Introduction

Consider a family $\mathcal{S}$ of geometric objects (called "sites") in the plane. The farthest-site Voronoi diagram of $\mathcal{S}$ subdivides the plane into regions, each region associated with one site $P \in \mathcal{S}$, and containing those points $x \in \mathbb{R}^2$ for which P is the farthest among the sites of $\mathcal{S}$.

While closest-site Voronoi diagrams have been studied extensively [1], their farthest-site cousins have received somewhat less attention. The case of (possibly intersecting) line segment sites was only solved recently by Aurenhammer et al. [2]; they gave an $O(n \log n)$ time algorithm to compute the diagram for n line segments.

Farthest-site Voronoi diagrams have a number of important applications. Perhaps the most well-known one is the problem of finding a smallest disk that intersects all the sites. This disk can be computed in linear time once the diagram is known, since its center is a vertex or lies on an edge of the diagram. Other applications are finding the largest gap to be bridged between sites, or building a data structure to quickly report the site farthest from a given query point.

We are here interested in the case of complex sites with non-constant description complexity. This setting was perhaps first considered by Abellanas et al. [3]:

* This research was supported by the French-Korean Science and Technology Amicable Relationships program (STAR), and the Brain Korea 21 Project, the School of Information Technology, KAIST, 2007.

** National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 407–418, 2007.

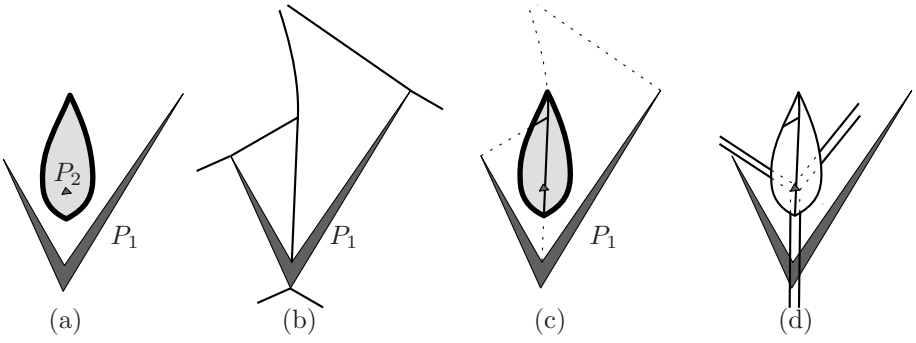


Fig. 1. (a) The bisector of two polygons can be a closed curve. (b) The medial axis $\mathfrak{M}(P_1)$. (c) The Voronoi region $\mathcal{R}(P_1)$. (d) The farthest-polygon Voronoi diagram $\mathfrak{F}(\{P_1, P_2\})$.

their sites are finite point sets, and so the distance to a site is the distance to the nearest point of that site. Put differently, they consider n points colored with k different colors, and their *farthest color Voronoi diagram* subdivides the plane depending on which color is farthest away. The motivation for this problem is the one mentioned above, namely to find a smallest disk that contains a point of each color—this is a facility location problem where the goal is to find a position that is as close as possible to each of k different types of facilities (such as schools, post offices, supermarkets, etc.). In a companion paper [4] the authors study other color-spanning objects.

The farthest color Voronoi diagram is easily seen to be the projection of the upper envelope of the k Voronoi surfaces corresponding to the k color classes. Huttenlocher et al. [5] show that this upper envelope has complexity $\Theta(nk)$ for n points, and can be computed in time $O(nk \log n)$ (see also the book by Sharir and Agarwal [6, Section 8.7]).

Van Kreveld and Schlechter [7] consider the farthest-site Voronoi diagram for a family of disjoint simple polygons. Again, they are interested in finding the center of the smallest disk intersecting or touching all polygons, which they then apply to the cartographic problem of labeling groups of islands. Their algorithm is based on the claim that this *farthest-polygon Voronoi diagram* is an instance of the *abstract farthest-site Voronoi diagram* defined by Mehlhorn et al. [8]—but this claim is false, since the bisector of two disjoint simple polygons can be a closed curve, see Fig. 1(a). In particular, Voronoi regions can be bounded (which is impossible for regions in abstract farthest-site Voronoi diagrams).

Note that the farthest-polygon Voronoi diagram can again be expressed as the upper envelope of k Voronoi surfaces—but this does not seem to lead to anything stronger than near-quadratic complexity and time bounds. We show in this paper that in fact the complexity of the farthest-polygon Voronoi diagram of k disjoint simple polygons of total complexity n is $O(n)$, independent of the number of polygons.

We give a divide-and-conquer algorithm with running time $O(n \log^3 n)$ to compute the farthest-polygon Voronoi diagram. Our key idea is to build point location data structures for the partial diagrams already computed, and to use parametric search on these data structures to find suitable starting vertices for the merging step. This idea may find applications to the computation of other complicated Voronoi diagrams. Our algorithm implies an $O(n \log^3 n)$ algorithm to compute the smallest disk touching or intersecting all the input polygons.

We note that for a family of disjoint *convex* polygons, finding the smallest disk touching all of them is much easier, and can be solved in time $O(n)$ (where n is the total complexity of the polygons) [9].

2 Definition of Farthest-Polygon Voronoi Diagrams

We consider a family $\mathcal{S}$ of k pairwise-disjoint polygonal sites of total complexity n. Here, a *polygonal site* of complexity m is the union of m line segments, whose interiors are pairwise disjoint, but whose union is connected. (In other words, the corners[1] and edges of a polygonal site form a one-dimensional connected simplicial complex in the plane.) In particular, the boundary of a simple polygon is a polygonal site and we can model a polygon with h holes using its $h+1$ boundaries and connecting them with at most h additional edges. For a point $x \in \mathbb{R}^2$, the distance $d(x, P)$ between x and a site $P \in \mathcal{S}$ is the distance from x to the closest point on P.

We assume that the family $\mathcal{S}$ is in general position, that is, no disk touches four edges, and no two edges are parallel.

The *features* of a site P are its corners and edges. For a site $P \in \mathcal{S}$, we define the function $\Psi_P : \mathbb{R}^2 \mapsto \mathbb{R}$ as $\Psi_P(x) = d(x, P)$. The graph of Ψ_P is a *Voronoi surface*, it is the lower envelope of circular cones for each corner of P and of rectangular wedges for each edge of P. The orthogonal projection of this surface on the plane induces a subdivision of the plane, the *medial axis* $\mathfrak{M}(P)$ of P. Each cell C of this subdivision corresponds to a feature w of P, it is the set of all points $x \in \mathbb{R}^2$ such that w is or contains the unique closest point on P to x. Here, edges of P are considered relatively open, so the cell of a corner is disjoint from the cells of its incident edges. The boundaries between cells are the *arcs* and *vertices* of the medial axis. Since the medial axis of P is the closest-site Voronoi diagram of P's features, it can be computed in time $O(m \log m)$, where m is the complexity of P [10].

A pocket of P is a connected component of $\mathrm{CH}(P) \setminus P$, where $\mathrm{CH}(P)$ is the convex hull of P. The medial axis $\mathfrak{M}(P)$ contains exactly one tree for each pocket of P. If a pocket shares an edge with $\mathrm{CH}(P)$ that is not an edge of P, then its medial axis tree contains exactly one infinite arc. (There can be additional infinite arcs for each corner of an edge of P that appear on the convex hull).

We now consider the function $\Phi : \mathbb{R}^2 \mapsto \mathbb{R}$ defined as $\Phi(x) = \max_{P \in \mathcal{S}} \Psi_P(x)$. The graph of Φ is the upper envelope of the surfaces Ψ_P, for $P \in \mathcal{S}$. The surface Φ consists of conical and planar patches from the Voronoi surfaces Ψ_P, and the

[1] We reserve the word "vertex" for vertices of the Voronoi diagram.

arcs separating such patches are either arcs of a Voronoi surface Ψ_P (we call these *medial axis arcs*), or intersection curves of two Voronoi surfaces Ψ_P and Ψ_Q (we call these *pure arcs*). The vertices of Φ are of one of the following three types:

- Vertices of one Voronoi surface Ψ_P. We call these *medial axis vertices*.
- Intersections of an arc of Ψ_P with a patch of another surface Ψ_Q. We call these *mixed vertices*.
- Intersections of patches of three Voronoi surfaces Ψ_P, Ψ_Q, Ψ_R. We call these *pure vertices*.

The projection of the surface Φ onto the plane is the *farthest-polygon Voronoi diagram* $\mathfrak{F}(\mathcal{S})$ of $\mathcal{S}$. It is a subdivision of the plane into cells, arcs, and vertices. For a point $x \in \mathbb{R}^2$, let us define $D(x) = D_{\mathcal{S}}(x)$ as the smallest disk centered at x that intersects all sites $P \in \mathcal{S}$. By definition, there is always at least one site that touches $D(x)$ without intersecting its interior, and the radius of $D(x)$ is equal to $\Phi(x)$. By our general position assumption, only the following five cases can occur:

- If $D(x)$ touches one site P in only one feature w, and all other sites intersect the interior of $D(x)$, then x lies in a cell of $\mathfrak{F}(\mathcal{S})$, and the cell belongs to the feature w of P.
- If $D(x)$ touches one site P in two or three features, and all other sites intersect the interior of $D(x)$, then x lies on a medial axis arc or medial axis vertex of $\mathfrak{F}(\mathcal{S})$, and is incident to cells belonging to different features of P.
- If $D(x)$ touches one feature w of site P, one feature u of site Q, and all other sites intersect the interior of $D(x)$, then x lies on a pure arc separating cells belonging to features w and u.
- If $D(x)$ touches two features of site P and one feature of site Q, and all other sites intersect the interior of $D(x)$, then x is a mixed vertex incident to a medial axis arc of P.
- If $D(x)$ touches one feature each of three sites P, Q, and R, and all other sites intersect the interior of $D(x)$, then x is a pure vertex.

Put differently, vertices of $\mathfrak{F}(\mathcal{S})$ are points $x \in \mathbb{R}^2$ where $D(x)$ touches three distinct features of sites. If all three features are on the same site, the vertex is a medial axis vertex. If the three features are on three distinct sites, then the vertex is a pure vertex. In the remaining case, if two features are on a site P, and the third feature is on a different site Q, the vertex is a mixed vertex.

Consider now an arc α of $\mathfrak{F}(\mathcal{S})$. If we let x move along α, then $\Phi(x)$—which is the radius of $D(x)$—changes continuously. Since the arc is defined by two features (corners or edges), $\Phi(x)$ cannot assume a local maximum in the interior of α, but it can assume a local minimum. We denote the location of such a local minimum a *pseudo-vertex* of $\mathfrak{F}(\mathcal{S})$. After introducing pseudo-vertices, $\Phi(x)$ is a monotone function on each arc, and we can thus orient all arcs in the direction of increasing $\Phi(x)$.

Fig. 2 illustrates all different vertex types, including the two types of pseudo-vertices of degree two (types (c) and (f)).

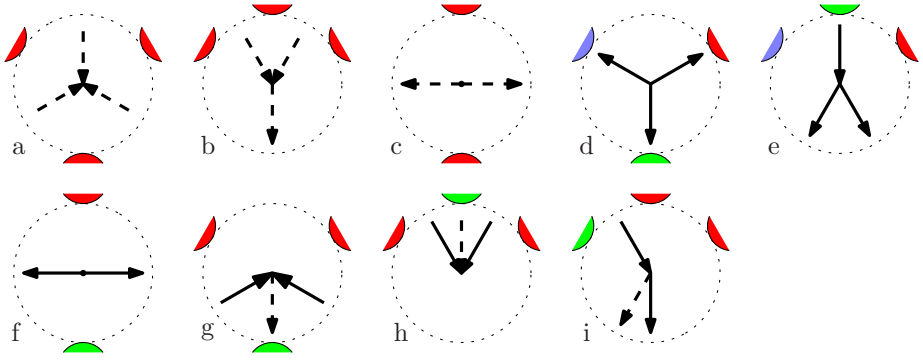


Fig. 2. The different types of vertices in the farthest-polygon Voronoi diagram. Pure arcs are shown solid, medial axis arcs dashed. The arrows indicate the direction of increasing $\Phi(x)$.

Since the local shape of $\mathfrak{F}(\mathcal{S})$ around a vertex v is determined solely by the features defining the vertex, in each configuration we can uniquely determine the orientation of the incident arcs. Fig. 2 shows the orientation of these arcs in each case. This orientation plays a crucial role in the next section, where we show that the complexity of the diagram, that is, the total number of vertices, is only $O(n)$.

In addition to the nine types of vertices discussed above, we need to consider *vertices at infinity*, that is, we consider the semi-infinite arcs of $\mathfrak{F}(\mathcal{S})$ to have a degree-one vertex at their end. For a vertex v at infinity, the "disk" $D(v)$ is a halfplane, and we have two cases:

- If $D(v)$ touches one site P in two features, and all other sites intersect the interior of $D(v)$, then $D(v)$ is the "infinite" endpoint of a medial axis arc, and we consider it a medial axis vertex at infinity.
- If $D(v)$ touches two distinct sites, and all other sites intersect its interior, then $D(v)$ is the "infinite" endpoint of a pure edge, and we consider it a pure vertex at infinity.

Finally, let us define the *Voronoi region* of a site $P \in \mathcal{S}$. The Voronoi region $\mathcal{R}(P)$ of P is simply the union of all cells, medial axis arcs, and medial axis vertices of $\mathfrak{F}(\mathcal{S})$ belonging to features of P. Voronoi regions are not necessarily connected, in fact, a single Voronoi region can have up to $k-1$ connected components.

3 Complexity Bound

Consider again Fig. 2. Let us call a vertex a *source* if it has more outgoing pure arcs than incoming pure arcs, and a *sink* if it has more incoming pure arcs than outgoing pure arcs. As shown in the figure, all finite pure vertices are sources.

The pure vertices at infinity are sinks. The only other sinks are the mixed vertices of type (g) and (h). Note that mixed vertices of type (i) are neither sources nor sinks.

We can now partition the pure arcs of $\mathfrak{F}(\mathcal{S})$ into disjoint directed paths. Each such path starts at a source, and ends at a sink. This implies that we can bound the number of sources by twice the number of sinks, as at most two paths can end in a sink.

We show in the following that the number of pure vertices at infinity is at most $2k-2$, and that the number of mixed vertices is $O(n)$. Since the total number of vertices of all medial axes $\mathfrak{M}(P)$, for $P \in \mathcal{S}$, is $O(n)$, this implies the main theorem of this section:

Theorem 1. *The complexity of the farthest-polygon Voronoi diagram of a family of disjoint polygonal sites of total complexity n is $O(n)$.*

We first discuss the number of vertices at infinity.

Lemma 1. *The number of pure vertices at infinity of $\mathfrak{F}(\mathcal{S})$ is at most $2k-2$. The total number of vertices at infinity of $\mathfrak{F}(\mathcal{S})$ is $O(n)$.*

Proof. For two sites $P, Q \in \mathcal{S}$, consider the diagram $\mathfrak{F}(\{P,Q\})$. A pure vertex at infinity corresponds to an edge of $\mathrm{CH}(P \cup Q)$ supported by a corner of P and a corner of Q. But $\mathrm{CH}(P \cup Q)$ can have at most two such edges, since P and Q are disjoint and both are connected, and so $\mathfrak{F}(\{P,Q\})$ has at most two pure vertices at infinity.

Consider now again $\mathfrak{F}(\mathcal{S})$, and let $\sigma(\mathcal{S})$ denote the sequence of sites whose Voronoi regions appear at infinity in circular order, starting and ending at the same region. We claim that $\sigma(\mathcal{S})$ is a Davenport-Schinzel sequence of order 2, and has therefore length at most $2k-1$ [6]. Indeed, $\sigma(\mathcal{S})$ has by definition no two consecutive identical symbols. Assume now that there are two sites P and Q such that the subsequence $PQPQ$ appears in $\sigma(\mathcal{S})$. If we delete all other sites, then $\sigma(\{P,Q\})$ would still need to contain the subsequence $PQPQ$, and therefore $\mathfrak{F}(\{P,Q\})$ would contain at least three pure vertices at infinity, a contradiction to the observation above.

It now suffices to observe that the pure vertices at infinity are exactly the transitions between consecutive Voronoi regions, and their number is at most $2k-2$. All remaining vertices at infinity are medial axis vertices. Since the total complexity of all $\mathfrak{M}(P)$, for $P \in \mathcal{S}$, is $O(n)$, the bound follows. □

It remains to show that the total number of mixed vertices is $O(n)$. Consider a site $P \in \mathcal{S}$ of complexity m. Its medial axis $\mathfrak{M}(P)$ has complexity $O(m)$, and we can consider $\mathfrak{M}(P)$ as a graph embedded in $\mathbb{R}^2 \setminus P$. It consists of a collection of trees. In Lemma 3 below we show that for each tree $\mathcal{T}$ of $\mathfrak{M}(P)$ the intersection $\mathcal{T} \cap \mathcal{R}(P)$ is a connected subtree. Since the mixed vertices on $\mathcal{T}$ are exactly the finite leaves of this subtree, this implies that the number of mixed vertices on $\mathfrak{M}(P)$ is $O(m)$. Summing over all $P \in \mathcal{S}$ then proves that the number of mixed vertices of $\mathfrak{F}(\mathcal{S})$ is $O(n)$.

It remains to prove Lemma 3. We need another notation: for a point $x \in \mathbb{R}^2$ and a site P, let $D_P(x)$ denote the largest disk centered at x whose interior does not intersect P (and which is therefore touching P). Note that a point $x \in \mathfrak{M}(P)$ is in $\mathcal{R}(P)$ if and only if $D_{\mathcal{S}}(x) = D_P(x)$, which is true if and only if all other sites intersect the interior of $D_P(x)$.

Lemma 2. *Let γ be a path in $\mathfrak{M}(P)$, let $Q \in \mathcal{S} \setminus \{P\}$ be another site, and let γ_Q be the set of points $x \in \gamma$ where $D_P(x)$ intersects Q. Then γ_Q is a connected subset of γ, that is, a subpath.*

Proof. We can assume γ to be a maximal path in $\mathcal{T}$, connecting a corner w of P with another corner u (or possibly going to infinity). Assume for a contradiction that there are points x, y, z on γ in this order such that $x, z \in \gamma_Q$, but $y \notin \gamma_Q$.

The disk $D_P(y)$ separates one connected component of $\mathbb{R}^2 \setminus P$ into two components A and B. The two endpoints of γ must lie in different components, say $w \in A$ and $u \in B$.

We first argue that any disk D that does not intersect P cannot contain points in both A and B. Indeed, the boundary of D would have to intersect the boundary of $D_P(y)$ in four points, a contradiction.

The disk $D_P(z)$ touches P on the boundary of B, and so it contains a point in B—which implies that it cannot contain a point in A. Since $D_P(y)$ does not intersect Q and Q is connected, this implies that Q is entirely contained in B. On the other hand, the disk $D_P(x)$ touches P on the boundary of A, and by our claim that means that it cannot intersect B. That implies that $D_P(x) \cap Q$ is empty, and the claim follows. □

The following lemma is an easy consequence of Lemma 2.

Lemma 3. *Let $\mathcal{T}$ be a tree of $\mathfrak{M}(P)$. Then $\mathcal{T} \cap \mathcal{R}(P)$ is a connected subtree of $\mathcal{T}$.*

4 Computing the Voronoi Diagram

The proof of Theorem 1 suggests an algorithm for computing the diagram by tracing the paths considered there. This is roughly equivalent to computing the surface Φ by sweeping a horizontal plane downwards, and maintaining the part of Φ above this plane. This is essentially the approach used by Aurenhammer et al. [2] for the computation of farthest-segment Voronoi diagrams. It does not seem to work for our diagram because of the mixed vertices of type (h), where Φ has a local maximum.

We instead offer a divide-and-conquer algorithm.

Theorem 2. *The farthest-polygon Voronoi diagram $\mathfrak{F}(\mathcal{S})$ of a family $\mathcal{S}$ of disjoint polygonal sites of total complexity n can be computed in expected time $O(n \log^3 n)$.*

Proof. Let $\mathcal{S} = \{P_1, \ldots, P_k\}$, and let n_i be the complexity of P_i. If $k = 1$, then $\mathfrak{F}(\mathcal{S})$ is simply the medial axis $\mathfrak{M}(P_1)$, which can be computed in time $O(n \log n)$ [10]. Otherwise, we split $\mathcal{S}$ into two disjoint families $\mathcal{S}_1, \mathcal{S}_2$ as follows:

- If there is a site P_i with complexity $n_i \geqslant n/2$, then $\mathcal{S}_1 = \{P_i\}$ and $\mathcal{S}_2 = \mathcal{S} \setminus \{P_i\}$.
- Otherwise there must be an index j such that $n/4 \leqslant \sum_{i=1}^{j} n_i \leqslant 3n/4$. We let $\mathcal{S}_1 = \{P_1, \ldots, P_j\}$ and $\mathcal{S}_2 = \{P_{j+1}, \ldots, P_k\}$.

We recursively compute $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$. We show below that we can then merge these two diagrams to obtain $\mathfrak{F}(\mathcal{S})$ in time $O(n \log^2 n)$, proving the theorem. □

It remains to discuss the merging step. We are given a family $\mathcal{S}$ of disjoint polygonal sites of total complexity n, and we are given $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$, where $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ is a disjoint partition of $\mathcal{S}$.

Consider the diagram $\mathfrak{F}(\mathcal{S})$ to be computed. We color the Voronoi regions of $\mathfrak{F}(\mathcal{S})$ defined by sites in $\mathcal{S}_1$ red, and the Voronoi regions defined by sites in $\mathcal{S}_2$ blue. A pure arc of $\mathfrak{F}(\mathcal{S})$ is red if it separates two red regions, and blue if it separates two blue regions. The remaining pure arcs, which separate a red and a blue region, are called *purple*. A vertex of $\mathfrak{F}(\mathcal{S})$ is purple if it is incident to a purple arc. We observe that by our general position assumption, every finite purple vertex is incident to exactly two purple arcs, and so the purple arcs form a collection of open and closed chains, see Fig. 2.

Merging $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$ can be done in linear time once all purple arcs are known. In fact, the diagram $\mathfrak{F}(\mathcal{S})$ consists of those portions of $\mathfrak{F}(\mathcal{S}_1)$ lying in the red regions of $\mathfrak{F}(\mathcal{S})$, and those portions of $\mathfrak{F}(\mathcal{S}_2)$ lying in the blue regions of $\mathfrak{F}(\mathcal{S})$.

We show below that if we have a starting vertex on every purple chain, then we can trace the purple chains in total time $O(n)$. For the open chains, we can use the purple vertices at infinity as starting vertices, as these are easy to compute. For the closed purple chains, we make use of the following lemma:

Lemma 4. *Any closed purple chain contains a mixed vertex.*

Proof. A closed purple chain is a compact set in the plane, and so $\Phi(x)$ assumes a maximum in some point x on this curve. But Φ can assume a local maximum only in mixed vertices of type (g) and (h), see Fig. 2. □

It remains to discuss the following three steps: (a) Computing the pure vertices at infinity; (b) Computing the mixed vertices; (c) Tracing the purple chains. Computing the pure vertices at infinity is relatively easy, making use of the same Davenport-Schinzel arguments used in Lemma 1, and we omit the details in this extended abstract.

4.1 Tracing the Purple Chains

We first need to discuss a monotonicity property of cells of $\mathfrak{F}(\mathcal{S})$. Let C be a cell of $\mathfrak{F}(\mathcal{S})$ belonging to feature w of site P. For a point $x \in C$, let x^* be the point on w closest to x. Let f_x be a directed line segment starting at x and extending in direction $\overrightarrow{x^*x}$ until we reach $\mathfrak{M}(P)$ (a semi- infinite segment if this does not happen). We call f_x the *fiber* of x. We note that if w is an edge, then all fibers of C are parallel and normal to w; if w is a corner then all fibers are supported by lines through w.

Lemma 5. *For any $x \in C$, the fiber f_x lies entirely in C (and therefore in $\mathcal{R}(P)$).*

Proof. The disk $D(x)$ touches P in x^* only, and its interior intersects all other sites. When we move a point y from x along f_x, the disk D centered at y through x^* keeps containing $D(x)$, and it therefore still intersects all other sites. This implies that $y \in C$ as long as D does not intersect P in another point. This does not happen until we reach $\mathfrak{M}(P)$. □

An immediate consequence is that cells are "monotone", this follows easily from Lemma 5.

Lemma 6. *Let C be a cell belonging to feature w. If w is a corner, then any line through w intersects C in a segment. If w is an edge, then any line normal to w intersects C in a segment.*

Lemma 6 implies that the boundary of a cell C belonging to a feature w consists of two chains monotone with respect to w (that is, monotone in the direction of an edge, and rotationally monotone around a corner). The *lower chain* is closer to the feature and consists of pure arcs only, the *upper chain* consists of medial axis arcs only.

An important consequence of Lemma 5 for tracing the purple chains is the following:

Lemma 7. *Let f_x be a fiber of a cell C in $\mathfrak{F}(\mathcal{S}_1)$ or $\mathfrak{F}(\mathcal{S}_2)$. Then f_x is intersected by at most one purple arc of $\mathfrak{F}(\mathcal{S})$.*

Proof. Assume for the contrary that two purple arcs intersect f_x in points p and q, where q lies on f_p. Then there is a point p' on f_x very close to p such that $p' \in \mathcal{R}(P)$ in $\mathfrak{F}(\mathcal{S})$, where C belongs to site P. But then the fiber $f_{p'}$ lies in $\mathcal{R}(P)$ in $\mathfrak{F}(\mathcal{S})$, and cannot intersect a purple arc at q. □

Lemma 7 allows us to trace purple chains very easily. Once we have located a starting vertex in both $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$, we trace the chain through both diagrams at the same time. We observe that every time it intersects a cell boundary in $\mathfrak{F}(\mathcal{S}_1)$ or $\mathfrak{F}(\mathcal{S}_2)$, we have indeed found a vertex of the purple chain, so the total number of such intersections is only $O(n)$.

When we enter a cell C along a purple arc, we simply follow the arc through the cell as it intersects the fibers of C. We consider the upper and lower chain of C at the same time, and trace the cells of $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$ in parallel. This allows us to charge the cost of tracing to those features of the two cells incident to the fibers the purple chain actually intersects. Since no fiber is intersected by more than one purple arc, the total tracing time is $O(n)$.

4.2 Computing the Mixed Vertices

This is the hardest part of the algorithm. We start by computing the randomized[2] point-location data structure of Mulmuley [12] (see also [13, Chapter 6]) for the

[2] This is the only use of randomization in our algorithm. It can probably be avoided by using the point location data structure of Edelsbrunner et al. [11] instead.

two given Voronoi diagrams $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$. This data structure only needs two primitive operations:

- For a given point p in the plane, determine whether the query point x lies left or right of p; and
- For an x-monotone line segment or parabola arc γ, determine whether the query point lies above or below γ.

Both cases can be summarized as follows: Given a *comparator* γ, determine on which side of γ the query point x lies. The comparator can be either a line or a parabola arc.

We compute the mixed vertices lying on each medial axis $\mathfrak{M}(P)$ separately. For each tree $\mathcal{T}$ of $\mathfrak{M}(P)$, the intersection $\mathcal{T} \cap \mathcal{R}(P)$ is a connected subtree by Lemma 3. We can determine the *inner* vertices of this subtree easily, by performing a point location operation for each vertex v of $\mathcal{T}$ in $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$. This tells us which site is farthest from v—and v lies in $\mathcal{R}(P)$ if and only if this is the site P. Let I be the set of vertices of $\mathcal{T}$ that lie in $\mathcal{R}(P)$. We now need to consider two cases.

If I is non-empty, then every arc α of $\mathcal{T}$ incident to one vertex in I and one vertex not in I must contain exactly one mixed vertex x^* by Lemma 3. We locate this vertex x^* by using *parametric search* along the arc α [14]. The idea is to execute two point location queries in $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$ using x^* as the query point. Each query executes a sequence of primitive operations, where we compare the (unknown) location of x^* with a comparator γ (a line or a parabola arc). This primitive operation can be implemented by intersecting α with γ, resulting in a set of at most four points. In $O(\log n)$ time, we can test for each of these points whether it lies in $\mathcal{R}(P)$. This tells us between which of these points the unknown mixed vertex x^* lies, and we can answer the primitive operation.

It follows that we can execute the two point location queries in time $O(\log^2 n)$, and we obtain the cells C_1 and C_2 of $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$ containing x^*. The mixed vertex x^* lies on the bisector of the features w_1 and w_2 to which these two cells belong (one of them is necessarily a feature of P). We compute the intersection of this bisector with α to obtain x^*.

It remains to consider the case where I is empty, that is, no vertex of $\mathcal{T}$ lies in $\mathcal{R}(P)$. Nevertheless, the region $\mathcal{R}(P)$ may intersect a single arc α of $\mathcal{T}$, and there are then two mixed vertices on α that we need to find. We first need to identify the arcs of $\mathcal{T}$ where this could happen.

Let p and q be two points on the same arc α of $\mathfrak{M}(P)$. We define the *cylinder* $\mathfrak{C}(p, q)$ of the pair (p, q) as

$$\mathfrak{C}(p, q) = \bigcup_{x \in pq} D_P(x) \setminus D_P(p),$$

where the union is taken over all points x on the arc α between p and q, see Fig. 3. We define a condition $G(p, q)$ as follows: Let Q be a site farthest from p, and let w be a feature of Q closest to p. Then $G(p, q)$ is true if $w \in \mathfrak{C}(p, q)$ or if Q intersects $D_P(q)$. We omit the easy proof of the following lemma:

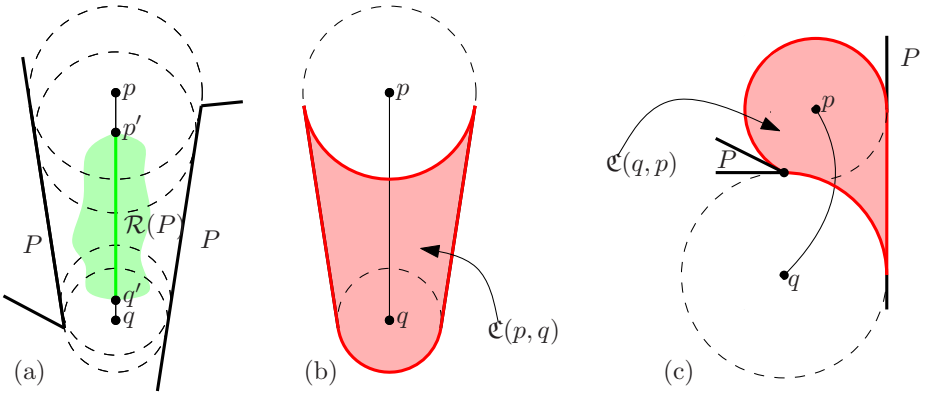


Fig. 3. (a) pq is a sub-arc of $\mathfrak{M}(P)$. $p'q'$ is the intersection of pq with $\mathcal{R}(P)$. We also have $p'q' = \mathfrak{M}(P) \cap \mathcal{R}(P)$. (b) The cylinder $\mathfrak{C}(p,q)$ of the pair (p,q) in (a). (c) The cylinder $\mathfrak{C}(q,p)$ of the pair (q,p) for a parabola arc.

Lemma 8. *Let p, q be points on the same arc α of $\mathfrak{M}(P)$, such that neither p nor q lie in $\mathcal{R}(P)$. If α intersects $\mathcal{R}(P)$ between p and q, then $G(p,q)$ and $G(q,p)$ both hold.*

Let us call an arc α connecting vertices p and q of $\mathcal{T}$ a *candidate arc* if $G(p,q)$ and $G(q,p)$ both hold.

Lemma 9. *If α is a candidate arc of a tree $\mathcal{T}$ of $\mathfrak{M}(P)$, then all points in $\mathcal{T} \cap \mathcal{R}(P)$ lie on α.*

Proof. Let α connecting p and q be a candidate arc. Since $G(p,q)$ holds, there is a site $Q \neq P$ such that Q does not intersect $D_{\mathcal{S}}(p)$, but does intersect $\mathfrak{C}(p,q)$. In particular, there must be a point $x \in \alpha$ such that Q intersects $D_{\mathcal{S}}(x) \supset D_P(x)$. Let now y be a point on $\mathcal{T}$ such that the path from q to y goes through α. By Lemma 2, Q cannot intersect $D_P(y)$, and so $y \notin \mathcal{R}(P)$.

Symmetrically, since $G(q,p)$ holds, we find that any point $z \in \mathcal{T}$ such that the path from p to z goes through α cannot be in $\mathcal{R}(P)$, and so any point of $\mathcal{T} \cap \mathcal{R}(P)$ must lie on α. □

Lemma 9 implies immediately that if there are two candidate arcs, then $\mathcal{T} \cap \mathcal{R}(P)$ is empty, and there are no mixed vertices on $\mathcal{T}$.

Since we have point-location data structures for $\mathfrak{F}(\mathcal{S}_1)$ and $\mathfrak{F}(\mathcal{S}_2)$, we can test the condition $G(p,q)$ in time $O(\log n)$ for a given arc α in $\mathfrak{M}(P)$ and two points $p, q \in \alpha$. This allows to identify all candidate arcs in $O(m \log n)$ time, where m is the complexity of $\mathcal{T}$. If there is zero or more than one candidate arcs, we can stop immediately, as there are no mixed vertices on $\mathcal{T}$.

We briefly sketch how to handle the case of the presence of a single candidate arc α. We apply parametric search guided by our predicate $G(\cdot,\cdot)$. This leads either to the conclusion that $\mathcal{R}(P)$ is empty, or to the discovery of a point inside

$\alpha \cap \mathcal{R}(P)$, which we can plug in the previous method (when I is not empty) to find the two mixed vertices on α.

Acknowledgments

We thank the participants of the 8th Korean Workshop on Computational Geometry, organized by Tetsuo Asano at JAIST, Kanazawa, Japan, Aug. 1–6, 2005.

References

1. Aurenhammer, F., Klein, R.: Voronoi diagrams. In: Sack, J.R., Urrutia, J.,(eds). Handbook of Computational Geometry. pp. 201–290. Elsevier Science Publishers B.V. North-Holland, Amsterdam (2000)
2. Aurenhammer, F., Drysdale, R.L.S., Krasser, H.: Farthest line segment Voronoi diagrams. Information Processing Letters 100, 220–225 (2006)
3. Abellanas, M., Hurtado, F., Icking, C., Klein, R., Langetepe, E., Ma, L., Palop, B., Sacristán, V.: The farthest color Voronoi diagram and related problems. In: Abstracts 17th European Workshop Comput. Geom., pp. 113–116. Freie Universität, Berlin (2001)
4. Abellanas, M., Hurtado, F., Icking, C., Klein, R., Langetepe, E., Ma, L., Palop, B., Sacristán, V.: Smallest color-spanning objects. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 278–289. Springer, Heidelberg (2001)
5. Huttenlocher, D.P., Kedem, K., Sharir, M.: The upper envelope of Voronoi surfaces and its applications. Discrete Comput. Geom. 9, 267–291 (1993)
6. Sharir, M., Agarwal, P.K.: Davenport-Schinzel Sequences and Their Geometric Applications. Cambridge University Press, New York (1995)
7. van Kreveld, M., Schlechter, T.: Automated label placement for groups of islands. In: Proc. of the 22nd International Cartographic Conference (2005)
8. Mehlhorn, K., Meiser, S., Rasch, R.: Furthest site abstract Voronoi diagrams. Int. J. Comput. Geom. & Appl. 11(6), 583–616 (2001)
9. Jadhav, S., Mukhopadhyay, A., Bhattacharya, B.K.: An optimal algorithm for the intersection radius of a set of convex polygons. J. Algorithms 20, 244–267 (1996)
10. Fortune, S.J.: A sweepline algorithm for Voronoi diagrams. Algorithmica 2, 153–174 (1987)
11. Edelsbrunner, H., Guibas, L.J., Stolfi, J.: Optimal point location in a monotone subdivision. SIAM Journal on Computing 15(2) (1986)
12. Mulmuley, K.: A fast planar partition algorithm, I. J. Symbolic Comput. 10(3-4), 253–280 (1990)
13. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications, 2nd edn. Springer, Berlin, Germany (2000)
14. Megiddo, N.: Applying parallel computation algorithms in the design of serial algorithms. J. ACM 30(4), 852–865 (1983)

Equitable Revisited

Wolfgang Bein[1,⋆], Lawrence L. Larmore[1,⋆], and John Noga[2]

[1] Center for the Advanced Study of Algorithms, School of Computer Science, University of Nevada Las Vegas, Nevada 89154, USA
bein@cs.unlv.edu, larmore@cs.unlv.edu
[2] Computer Science Department, California State University, Northridge, CA 91330-8281, USA
jnoga@csun.edu

Abstract. The randomized k-paging algorithm EQUITABLE given by Achlioptas *et al.* is H_k-competitive and uses $O(k^2 \log k)$ memory. This competitive ratio is best possible. The randomized algorithm RMARK given by Fiat *et al.* is $(2H_k - 1)$-competitive, but only uses $O(k)$ memory. Borodin and El Yaniv [6] list as an open question whether there exists an H_k-competitive randomized algorithm which requires $O(k)$ memory for k-paging. In this paper we answer this question in the affirmative by giving a modification of EQUITABLE.

Keywords: Design of Algorithms, Online Algorithms, Paging, Randomization.

1 Introduction

In the k-paging problem we consider a two-level memory system, consisting of fast memory, also referred as the *cache*, which can hold k memory units commonly called *pages*, and an area of slow memory capable of holding a much larger number of pages. If a page is needed in fast memory this is called a *page request*. Such a request causes a *hit* if the page is already in the cache at the time of the request. In the case of a *miss* (*i.e.* when the page is not in the cache) the requested page must be brought into the cache while a page in the cache must be evicted to make room for the new page. We assume that the total cost of copying a new page and ejecting a page is one, and all other costs are zero. Thus, given a sequence of requests, ϱ, the cost of an algorithm $\mathcal{A}$ on ϱ will be denoted $cost_{\mathcal{A}}(\varrho)$ and is simply the number of misses algorithm $\mathcal{A}$ incurs on ϱ. An online paging algorithm must make decisions about such evictions as the request sequence of pages is presented to the algorithm. Online algorithms are analyzed in terms of *competitiveness*, a measure of the performance that compares the decision made online with the optimal offline solution for the same problem, where the lowest

⋆ Research of these authors supported by NSF grant CCR-0312093 and UNLV sabbatical leave (Bein).

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 419–426, 2007.

possible competitiveness is best. We say that a randomized online algorithm $\mathcal{A}$ for the k-paging problem is *C-competitive* if there is a constant K such that, given any request sequence ϱ, $cost_{\mathcal{A}}(\varrho) \leq C \cdot cost_{opt}(\varrho) + K$, where $cost_{opt}(\varrho)$ is the optimal cost for sequence ϱ and $cost_{\mathcal{A}}(\varrho)$ is the (expected) cost of the randomized algorithm.

For paging the optimal cost can be computed using Belady's algorithm [5], which replaces the page whose next request is latest. We also refer to this algorithm as OPT. A closely related concept is that of a work function which, when given a configuration, returns the minimum cost of serving the request sequence and ending in this configuration. In the case of paging a configuration is simply the set of k pages which are in the cache. Koutsoupias and Papadimitriou [8,9] give a complete characterization of the work function for paging.

We note that the competitive ratio is one aspect of an online algorithm, the memory requirement is another. It is typical to measure the memory requirement of an algorithm by the maximum number of pages remembered by the algorithm: any paging algorithm knows which k pages are currently in the cache and beyond that, an algorithm might store information about pages which are not currently in the cache. Those pages are called *bookmarks*. Thus the total memory requirement of a paging algorithm is k plus the number of bookmarks. We mention that if an online algorithm for the problem permits no bookmarks, it is called *trackless*, see [3].

It is well known that the deterministic competitiveness of the k-paging problem is k, which can be achieved by the trackless algorithm LRU [11]. The randomized competitiveness of the k-paging problem is known to be $H_k = \sum_{i=1}^{k} 1/i$. The lower bound result is due to [7]. The same paper also gives a $(2H_k - 1)$-competitive algorithm called RMark which uses only $O(k)$ memory; this algorithm is also trackless. An algorithm with best possible competitive ratio of H_k was first described in [10]; however their algorithm, Partition, is unbounded in its memory requirement. Later, [1] constructed another algorithm, Equitable, with best possible ratio H_k and a bounded memory requirement of $O(k^2 \log k)$ pages. The technique of decreasing memory used in [1] is called *forgiveness*. Under certain conditions, the algorithm simply assumes that OPT's cache is in a given configuration, despite the fact that it may not be. Informally speaking, the trick is that by this time the algorithm has enough "savings" to cover the cost of this "mistake." On the other hand Bein and Larmore [3] have shown that it is not possible for a trackless algorithm to achieve H_k-competitiveness if $k = 2$. Borodin and El-Yaniv list in [6] the open question whether there exists an H_k-competitive randomized algorithm which requires only $O(k)$ memory.

In this paper we answer this question in the affirmative. In the next section we review Equitable as needed for our paper. In section 3 we improve this technique so that the algorithm (still a version of Equitable) makes a forgiveness step sooner, and is never required to remember more than $3k$ pages. The basic idea is to not just forgive to one configuration but to use a rather more elaborate scheme.

2 Equitable

We assume that the reader is familiar with the definitions and notation of [1] although some definitions and results of that paper will be stated here for clarity and self-containedness. We remind the reader that many randomized online algorithms are given in a so-called distributional form, as is the algorithm Equitable of [1]. For randomized online algorithms against an oblivious adversary, the distribution model is equivalent to the more standard behavioral model (see *e.g.* [6]). For the k-cache problem, a *distributional algorithm* is essentially a state transition diagram, where each state is a probabilistic distribution of configurations. More precisely, a *configuration* is an unordered k-tuple of pages, which represents a possible cache configuration, and we denote by $\mathcal{S}^k$ be the set of all possible cache configurations. A transition from one state to the next is a deterministic transition to a new distribution. The cost of a move can be defined by the transport distance between the distributions.

In analyzing the competitiveness of online algorithms it is often useful to know the optimal solution for each request sequence. To this end, the *work function*, $\omega^\varrho : \mathcal{S}^k \to \mathbb{N}$, associated with a sequence of requests ϱ is defined as follows: $\omega^\varrho(S)$ is the minimum cost of servicing ϱ while ending in S. Note that if the last request r does not belong to S one can define $\omega^\varrho(S) = 1+\min_{S\in\mathcal{S}^k} \omega^\varrho(S+r-x)$. Note also that the optimal cost of servicing request sequence ϱ is $\min(\omega^\varrho)$. The superscript ϱ is usually dropped if it is clear from the context.

Given work function ω and request r, one can obtain the work function after service of request r – denoted as $(\omega\wedge r)$ – in the following way: $(\omega\wedge r)(S) = 1 + \min_{S\in\mathcal{S}^k} \omega^t(S + r - x)$ if $r \notin S$ and simply $(\omega\wedge r)(S) = \omega(S)$ if $r \in S$.

Let $\mathcal{T}$ be a set of configurations. We say that a work function is coned-up from $\mathcal{T}$, if for every configuration S there is a $T \in \mathcal{T}$ such that $\omega(S) = \omega(T) + ||S, T||$, where the last term denotes the cost of changing the cache S to the cache T, namely the number of pages in S which are not in T. If $\mathcal{T}$ is a singleton, we call ω a *cone*.

In this paper (as in [1]) it is convenient to normalize work functions in the following way: If $\min_{S\in\mathcal{S}^k} \{\omega(S)\} = \mathit{offset} > 0$ we lower the function by subtracting the constant *offset* and then storing that value; such functions are called *offset functions*. A notation system for offset functions was introduced by Koutsoupias and Papadimitriou [8,9]; we briefly summarize that concept here. To define any offset function, suppose $\emptyset = S_0 \subset S_1 \subset S_2 \subset \cdots \subset S_k$ are sets of pages, and let $m_i = |S_i|$. An offset function ω is then defined as follows:

1. We say that $S \in \mathcal{S}^k$ is in the *support* of ω, which we call $\mathit{supp}(\omega)$, if and only if $|S \cap S_i| \geq i$ for all $1 \leq i \leq k$. We have $\omega(S) = 0$ for all $S \in \mathit{supp}(\omega)$.
2. For any $T \in \mathcal{S}^k$, $\omega(T) = \min_{S\in \mathit{supp}(\omega)} ||S, T||$.

For convenience we also define $L_i = S_i - S_{i-1}$, called *layers*. We write either $S_1|S_2|\ldots|S_k|$ or equivalently $L_1|L_2|\ldots|L_k|$ to represent an offset function. If the initial set of pages in the cache is $\{s_1, s_2, \ldots, s_k\}$ then $S_i = \{s_1, s_2, \ldots, s_i\}$. Given an offset function $S_1|S_2|\ldots|S_k|$ the offset function resulting from a request to a page r is

$$\begin{array}{ll} \{r\}|S_2+r|S_3+r|\ldots|S_k+r| & \text{if } r \notin S_k, \\ \{r\}|S_1+r|S_2+r|\ldots|S_{i-1}+r|S_{i+1}|S_{i+2}|\ldots|S_k| & \text{if } r \in S_i \text{ and } i<k, \\ \{r\}|S_1+r|S_2+r|\ldots|S_{k-1}+r| & \text{if } r \in S_k, \end{array}$$

where the offset is increased by one in the first case. We will call requests of the first type *new* requests and requests of the second type *lazy*. We refer to S_k as the set of *active pages*. Thus, any algorithm which is based upon the offset function must use at least $m_k - k$ bookmarks, and we say that the algorithm has to "keep track of" these pages.

The concept of *forgiveness* is defined as replacing the current work function ω with another function ω' where $\omega(S) \geq \omega'(S)$ for all $S \in \mathcal{S}^k$. Note that if the algorithm makes a forgiveness step, the resulting function is no longer an accurate calculation of optimal cost but instead an underestimate. By a slight abuse of terminology we also refer to this estimate as an offset function.

As mentioned above, the algorithm EQUITABLE is defined using the distributional model. It is a *stable distribution* algorithm, *i.e.*, its distribution depends only on the offset function and not on any other information that could be computed from the sequence of requests thus far. If ω is the current offset function, the distribution π^ω will be by the procedure below. (We will omit the superscript ω from functions when the current offset function is clear from context).

1. For any $S \notin \mathit{supp}(\omega)$, $\pi(S) = 0$.
2. For any $S \in \mathit{supp}(\omega)$, $\pi(S) > 0$. The following random process selects a member of $\mathit{supp}(\omega)$:
   - Initialize S to be the empty set.
   - Let $T = S_k$.
   - Execute the following loop until $|S| = k$:
     (a) Select $x \in T$ uniformly at random.
     (b) Delete x from T.
     (c) If $S + x$ is a subset of any member of $\mathit{supp}(\omega)$ let $S = S + x$.
   - For any $S \in \mathit{supp}(\omega)$, define $\pi(S)$ to be the probability that S is selected in the previous procedure.

For an offset function ω and page x, define p_x^ω as the probability that the page x is in the set S selected in the previous procedure. Equivalently let $p_x = \sum_{\{S \in S^k | x \in S\}} \pi(S)$ which is the probability that x is in the cache. Note that if $p_x > 0$ then $x \in S_k$. Proofs of the following observations can be found in [1].

Observation 1. *If $x \in L_i$ and $y \in L_j$, where $i \leq j$, then $p_x \geq p_y$. If $i = j$ then $p_x = p_y$.*

Observation 2. *For any offset function, all sequences of lazy requests ending when $|S_k| = k$ have the same cost for* EQUITABLE.

We define the function Φ as the cost EQUITABLE incurs on a sequence of lazy requests ending when $|S_k| = k$.

3 Forgiveness Revisited

For any given offset function the algorithm EQUITABLE2 will maintain the same distribution as EQUITABLE. The difference between the algorithms is the forgiveness step. If $|S_k| = 3k$ and $r \notin S_k$ is requested then EQUITABLE2 moves to offset function $\{r\}|L_1|L_2|\ldots|L_{k-1}|$ and charges OPT zero. One way to view this step is that the requested page is moved into L_k prior to being requested. The motivation for this choice is that when the set of active pages is large, a request in L_k and a request outside S_k have nearly the same cost to EQUITABLE (if $|S_k| = 3k$ then a page in L_k will have cost between 2/3 and 1). If we can show that the small additional cost to the online algorithm can be covered, why not move the requested page into L_k, have it requested, and thereby reduce the size of the set of active pages?

Most of the notation of [1] is used, but the definition of Ψ is different. Recall that $m_i = |S_i|$ and define $\Psi = \Psi(\omega)$ as follows:

$$\Psi = \sum_{i=2}^{k} \Big(\frac{m_i}{i} + H_{i-1} - H_{m_i-1} - 1 \Big).$$

Similar to [1] we wish to show that for every request

$$cost + \Delta\Phi + \Delta\Psi \le H_k \cdot cost_{opt}$$

where *cost* denotes the expected cost to the algorithm at that step and $cost_{opt}$ is the optimal cost at that step.

In the discussion below, unprimed variables denote the values before a given request. Primed variables are the values after that request. Recall that $m_i \ge i$.

Lemma 1. *On a lazy request $r \in L_j$, $cost + \Delta\Phi + \Delta\Psi \le H_k \cdot cost_{opt}$.*

Proof. Since Φ is the cost for EQUITABLE to serve a lazy sequence of requests ending in a cone, on a lazy request $cost + \Delta\Phi = 0$. For a lazy request $cost_{opt} = 0$ by definition. So it suffices to show that $\Delta\Psi < 0$. We have

$$\begin{aligned}
\Delta\Psi &= \sum_{i=2}^{k} \left(\frac{m_i'}{i} - H_{m_i'-1} - \frac{m_i}{i} + H_{m_i-1} \right) \\
&= \sum_{i=2}^{j} \left(\frac{m_{i-1} + 1 - m_i}{i} - H_{m_{i-1}} + H_{m_i-1} \right) \\
&= \sum_{i=2}^{j} \left(\frac{m_{i-1} + 1 - m_i}{i} + \sum_{\ell=m_{i-1}+1}^{m_i-1} \frac{1}{\ell} \right) \\
&\le \sum_{i=2}^{j} \left(\frac{m_{i-1} + 1 - m_i}{i} + \sum_{\ell=m_{i-1}+1}^{m_i-1} \frac{1}{i} \right) \\
&= 0.
\end{aligned}$$

Lemma 2. *On a request $r \notin S_k$, if forgiveness does not occur, then $\Delta\Phi \leq \sum_{i=2}^{k} \frac{1}{m_i}$.*

Proof. If $k = 1$ then $\Phi = \Phi' = 0$ which shows that the lemma is true for $k = 1$. Assume $k > 1$ and that the lemma is true for $k - 1$.

For every page $s \neq r$, $p_s \geq p'_s$. Since $p_r = 0$ and $p'_r = 1$, $\sum_{s \in S_k} p_s - p'_s = 1$, there must be an item $x \in S_k$ for which $p_x - p'_x \leq 1/m_k$. Without loss of generality, $x \notin S_1$ because every item $y \in S_2$ will have $p_y - p'_y \leq p_x - p'_x$. Let this page $x \in S_j$ be the first item in the lazy request sequence which defines Φ and Φ'. Define the following offset functions:

$$\begin{aligned}
\omega &= S_1|S_2|\dots|S_k| \\
\omega' &= r|S_2 + r|\dots|S_k + r| \\
\omega \wedge x &= x|S_1 + x|S_2 + x|\dots|S_{j-1} + x|S_{j+1}|\dots|S_k| \\
\omega' \wedge x &= x|xr|S_2 + r + x|\dots|S_{j-1} + r + x|S_{j+1} + r|\dots|S_k + r| \\
\omega_{dropx} &= S_1|S_2|\dots|S_{j-1}|S_{j+1}|\dots|S_k| \\
\omega'_{dropx} &= r|S_2 + r|\dots|S_{j-1} + r|S_{j+1} + r|\dots|S_k + r|.
\end{aligned}$$

Now we notice that

$$\begin{aligned}
\Delta\Phi &\leq \frac{1}{m_k} + \Phi(\omega' \wedge x) - \Phi(\omega \wedge x) \\
&= \frac{1}{m_k} + \Phi(\omega'_{dropx}) - \Phi(\omega_{dropx}) \\
&\leq \frac{1}{m_k} + \sum_{i=2}^{k-1} \frac{1}{m_i} \\
&= \sum_{i=2}^{k} \frac{1}{m_i}
\end{aligned}$$

where the third line follows from the inductive hypothesis.

Lemma 3. *On a request $r \notin S_k$, if forgiveness does not occur, then*

$$cost + \Delta\Phi + \Delta\Psi \leq H_k \cdot cost_{opt}.$$

Proof. Since $r \notin S_k$, $m'_i = m'_i + 1$. Given lemma 2, it follows that

$$\begin{aligned}
cost + \Delta\Phi + \Delta\Psi &\leq 1 + \sum_{i=2}^{k} \frac{1}{m_i} + \sum_{i=2}^{k} \left(\frac{m'_i}{i} - H_{m'_i - 1} - \frac{m_i}{i} + H_{m_i - 1} \right) \\
&= 1 + \sum_{i=2}^{k} \frac{1}{m_i} + \sum_{i=2}^{k} \left(\frac{m_i + 1}{i} - H_{m_i} - \frac{m_i}{i} + H_{m_i - 1} \right) \\
&= \sum_{i=1}^{k} \frac{1}{i} \\
&= H_k \cdot cost_{opt}.
\end{aligned}$$

Lemma 4. *On a request outside S_k when forgiveness occurs.*

$$cost + \Delta\Phi + \Delta\Psi \leq 0.$$

Proof. A forgiveness step occurs when $S_k = 3k$ and a page $r \notin S_k$ is requested. The forgiveness step can be seen as placing r in L_k and then requesting r. We can easily compute $\Delta\Psi$ during a forgiveness step. However, it is easier to compute $cost + \Delta\Phi$ when r is added to L_k and on the request separately.

When r is placed into L_k the distribution must be adjusted and the lazy potential changes. The transportation cost necessary to adjust the distribution is p'_r. Since the cost on all lazy sequences is the same, we can compute the change in potential by considering the sequence which begins with a page $x \in L_k$. From Observation 1 $cost + \Delta\Phi = p_x - p'_x + p'_r = p_x \leq k/m_k$.

When r is requested $cost + \Delta\Phi = 0$ because Φ is the lazy potential. So it suffices to show that $k/m_k + \Delta\Psi$ is no more than 0. We have

$$\begin{aligned}
cost + \Delta\Phi + \Delta\Psi &\leq \frac{k}{m_k} + \sum_{i=2}^{k} \left(\frac{m'_i}{i} - H_{m'_i - 1} - \frac{m_i}{i} + H_{m_i - 1} \right) \\
&= \frac{1}{3} + \sum_{i=2}^{k} \left(\frac{m_{i-1} + 1 - m_i}{i} - H_{m_{i-1}} + H_{m_i - 1} \right) \\
&\leq \frac{1}{3} + \left(\frac{k - m_k}{k} - H_{k-1} + H_{m_k - 1} \right) \\
&\leq -\frac{5}{3} + H_{3k-1} - H_{k-1} \\
&\leq 0.
\end{aligned}$$

The second inequality above holds because the $\Delta\Psi$ is decreasing in m_ℓ for all $\ell < k$. So the worst case occurs when $m_\ell = \ell$.

Theorem 1. EQUITABLE2 *is an H_k-competitive, $O(k)$ memory, randomized algorithm for the k-paging problem.*

Proof. Since we perform a forgiveness step when the set of active pages has size $3k$ and a new page is requested, the set of pages we keep track of is never greater than $3k$. Lemmas 1, 3, and 4 show that $cost + \Delta\Phi + \Delta\Psi \leq H_k \cdot cost_{opt}$ on every request type. Noting that $\Phi + \Psi$ is initially 0 and never negative, summing over every request shows that $\text{EQUITABLE}(\varrho) \leq H_k \cdot \text{OPT}(\varrho)$ for any request sequence ϱ.

4 Conclusion

We have given an H_k-competitive randomized online algorithm for the k-paging problem which keeps track of only $3k$ pages. For large k, Lemma 4 can be improved to αk where $\alpha \approx 2.2572$ satisfies $\alpha^2 - \alpha - \alpha \log(\alpha) = 1$.

Bein *et al.* [2,4] show (using a different method than is used here) that for the 2-paging problem three pages suffice. For $k = 3$, the technique used in this paper can be used to calculate that 7 pages suffice. We also note that we have not proven $3k$ to be the minimum number of pages needed for any particular k. In fact, we conjecture that a stronger result holds than the upper bound from this paper:

Conjecture 1. There exists a randomized online algorithm for paging which is H_k-competitive and uses $o(k)$ bookmarks (i.e. $k + o(k)$ memory).

References

1. Achlioptas, D., Chrobak, M., Noga, J.: Competitive analysis of randomized paging algorithms. Theoret. Comput. Sci. 234, 203–218 (2000)
2. Bein, W., Fleischer, R., Larmore, L.L.: Limited bookmark randomized online algorithms for the paging problem. Inform. Process. Lett. 76, 155–162 (2000)
3. Bein, W., Larmore, L.L.: Trackless online algorithms for the server problem. Inform. Process. Lett. 74, 73–79 (2000)
4. Bein, W., Larmore, L.L., Reischuk, R.: Knowledge state algorithms: Randomization with limited information(2007) Arxiv: archive.org/cs/0701142
5. Belady, L.A.: A study of replacement algorithms for virtual storage computers. IBM Syst. J. 5, 78–101 (1966)
6. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press, Cambridge (1998)
7. Fiat, A., Karp, R., Luby, M., McGeoch, L.A., Sleator, D., Young, N.E.: Competitive paging algorithms. J. Algorithms 12, 685–699 (1991)
8. Koutsoupias, E., Papadimitriou, C.: Beyond competitive analysis. In: Proc. 35th Symp. Foundations of Computer Science (FOCS), pp. 394–400. IEEE, Los Alamitos (1994)
9. Koutsoupias, E., Papadimitriou, C.: Beyond competitive analysis. SIAM J. Comput. 30, 300–317 (2000)
10. McGeoch, L., Sleator, D.: A strongly competitive randomized paging algorithm. Algorithmica 6(6), 816–825 (1991)
11. Sleator, D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM 28, 202–208 (1985)

Online Scheduling of Equal-Length Jobs on Parallel Machines

Jihuan Ding[1,2], Tomáš Ebenlendr[3], Jiří Sgall[3], and Guochuan Zhang[1]

[1] Dept. of Mathematics, Zhejiang Univ., Hangzhou 310027, China
{dingjh,zgc}@zju.edu.cn
[2] College of Operations Research and Management Science, Qufu Normal Univ., Rizhao 276826, China
dingjihuan@hotmail.com
[3] Institute of Mathematics, AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic
{ebik,sgall}@math.cas.cz

Abstract. We study on-line scheduling of equal-length jobs on parallel machines. Our main result is an algorithm with competitive ratio decreasing to $e/(e-1) \approx 1.58$ as the number of machine increases. For $m \geq 3$, this is the first algorithm better than 2-competitive greedy algorithm.

Our algorithm has an additional property called *immediate decision*: at each time, it is immediately decided for each newly released job if it will be scheduled, and if so, then also the time interval and machine where it is scheduled is fixed and cannot be changed later. We show that for two machines, no deterministic algorithm with immediate decision is better than 1.8-competitive; this lower bound shows that our algorithm is optimal for $m = 2$ in this restricted model. We give some additional lower bounds for algorithms with immediate decision.

1 Introduction

We study a problem in the area of real-time scheduling. We are given an input sequence of jobs with equal processing times p. Each job has its release time and deadline, specifying the time window in which it needs to be scheduled; all the parameters are integers. The desired output is a nonpreemptive schedule on m identical machines, possibly only for a subset of the jobs on input. Each scheduled job must be executed between its release time and deadline, and different jobs cannot overlap if they are scheduled on the same machine. The term "nonpreemptive" means that each job must be executed without interruptions, in a contiguous interval of length p. The objective is to maximize the number of completed jobs.

In the *online version*, each job is released at its release time, and its deadline is revealed at this time, too. The number of jobs and future release times are unknown. At each time step when some machine is available (it just completed a job or was idle), we have to decide whether to start some jobs, and if so, to choose which ones, based only on the information about the jobs released so far.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 427–438, 2007.

An online algorithm is called *c-competitive* if on every input instance it schedules at least $1/c$ as many jobs as the optimum schedule.

This type of problems is extensively studied. In addition to the results mentioned below, there is a vast literature on real-time scheduling of jobs with arbitrary (not equal) processing times, weights, and so on. In these more general variants, often preemption is necessary to achieve reasonable competitive ratio, and still the ratio depends on various parameters.

For our problem with equal jobs and non-preemptive scheduling, it is known that a greedy algorithm performs reasonably well, namely it is 2-competitive for any number of machines. Note that in this type of problems it is usual and natural that more machines make performance better, as it is possible to keep in some time interval some fraction of the total capacity of the machines available for later arriving jobs. Nevertheless, for this very basic problem, no non-trivial algorithm was known for $m \geq 3$ before this work.

1.1 Previous Results

Most of the previous results deal only with the case of a single machine. In this case, it is known that a greedy algorithm is 2-competitive for this problem [2], and that this is optimal for deterministic algorithms [5]. There exists a 5/3-competitive randomized algorithm [3] and no better than 4/3-competitive randomized algorithm exists [5].

For two machines, independently Goldwasser and Pedigo [7] and Ding and Zhang [4] designed 1.5-competitive deterministic algorithms, and this competitive ratio is optimal. Their algorithms and analysis are fairly complicated, and it seems that this is necessary, as very precise timing is needed to obtain the optimal competitive ratio.

For more machines, 2-competitive greedy algorithm is known, which is an easy generalization of the single machine result. To our best knowledge, no better algorithms for $m \geq 3$, deterministic or randomized, were known prior to our work. A lower bound of 6/5 is known [4], using the same idea as the lower bound of 4/3 for randomized algorithms on a single machine. For deterministic algorithms, it actually gives a slightly higher lower bound, but it still approaches 6/5 for large m.

In the offline case, it is known that the problem is polynomially solvable for any fixed number of machines [1]. The complexity with the number of machines as the part of the input is still open.

1.2 Our Results

Our main result is algorithm BestFit with competitive ratio decreasing to $e/(e-1) \approx 1.582$ for large m. The exact ratio for m machines is

$$R_m = \frac{1}{1 - \left(\frac{m}{m+1}\right)^m}.$$

For $m = 2$, this competitive ratio is 1.8. Compared to the known algorithms for $m = 2$, both the algorithm and its analysis are very simple; the proof uses a charging scheme extending the analysis of the greedy algorithm for $m = 1$.

Immediate decision. Our algorithm has an additional property introduced in [4] which is called *immediate decision*: At each time t, it is immediately decided for each newly released job j (i.e., with $r_j = t$) if it will be scheduled, and if so, then also the time interval and machine where it is scheduled are fixed and cannot be changed later.

The second main contribution of our paper is a study of the power of the restricted model with immediate decision. We give several lower bounds showing that this restriction is quite severe.

For $m = 2$, we show that no deterministic algorithm with immediate decision is better than 1.8-competitive. Thus our algorithm BestFit is optimal for $m = 2$ in this restricted model. This shows that immediate decision is a strong requirement, as for $m = 2$ the optimal competitive ratio in the unrestricted model is 1.5 smaller.

We also give a simple lower bound of 4/3 for arbitrary m with immediate decision, which holds even for randomized algorithms.

Finally, we give some lower bounds for jobs with small processing times. With immediate decision for $m = 2$, we prove that deterministic algorithms are at least 1.6-competitive for $p = 1$ and 5/3-competitive for $p = 2$. The previously mentioned lower bound of 1.8 holds for any $p \geq 3$.

Scheduling with immediate decision is somewhat similar to the model of jobs arriving one by one, also called list scheduling, with an additional restriction that the jobs are ordered according to their release times. The models are the same for instances with all the release times distinct. In a model with continuous time, it is possible to slightly perturb the release times and deadlines, and the models are thus equivalent. For large p, our model with discrete time approaches the continuous time model and thus also the model of jobs arriving one by one consistently with the release times.

The requirement of immediate decision should also be compared with the notion of *immediate notification* introduced in [6]. Immediate notification requires that upon release of each job, we immediately decide (i.e., notify the user) if the job will be completed or not; however, its exact timing and the choice of a machine may depend on subsequent jobs. It seems to be the case that in all cases studied so far, it is possible to provide immediate notification without changing the performance of the algorithms. In contrast, our results show that this is no longer the case when the requirement is strengthened from immediate notification to immediate decision. Then the competitive ratio must increase at least in the case of two machines.

1.3 Preliminaries

Each job is described by a pair of numbers (r_j, d_j) denoting its release time and deadline, respectively. All the release times, deadlines and starting times

of jobs are assumed integral, as is usual in scheduling literature. Saying that a job is running at time t is equivalent to saying that it is running in time interval $[t, t+1)$. (However, we note that our algorithm and proofs work also for continuous time and non-integral r_j, d_j. The choice of the discrete model is mainly a matter of taste and tradition in the literature).

Given a particular schedule, S_j denotes the starting time of job j. Each scheduled job needs to be scheduled so that $r_j \leq S_j \leq d_j - p$. Let $C(M_i)$ denote the completion time of machine M_i, i.e., the first time t such that no job is scheduled to run on machine M_i at or after t.

In the online problem, each job is released at time r_j. However, in the more complicated lower bounds using adversary arguments, it is often convenient to say that at time t, based on the previous actions of the algorithm, we release some set of jobs, possibly including (r_j, d_j) with $r_j > t$. This should be interpreted so that we commit to releasing such job(s) and we actually reveal them to the algorithm only at time r_j.

2 The Algorithm

We now present our algorithm. It simply tries to schedule each job without any idle time on the most full machine where it can be completed by its deadline. So, for example, a sequence of jobs with huge deadlines is scheduled on a single machine, leaving all other machines available for future jobs.

Algorithm BestFit**.**
At each time t, as an invariant, each machine M_i is committed to execute jobs from t until its completion time $C(M_i)$ with no idle time inserted. A machine M_i is called *feasible* for job j if $C(M_i) \leq d_j - p$.

At time t, consider the newly released jobs one by one. If no feasible machine for job j exists, the job is rejected. Otherwise j is scheduled on a machine with largest $C(M_i)$, among all the feasible machines; job j is then scheduled to start execution at time $C(M_i)$.

For the analysis it is important to keep track of the order in which the jobs have been considered by the algorithm. We refer to this moment as to the *decision*. So, we have a linear ordering of jobs given by which job was decided earlier, which extends the ordering by release times. Also, a schedule at the time when a job was decided refers to the schedule at its release time, just before BestFit processed this job and fixed its schedule. Thus this schedule contains exactly all the jobs decided before the current one, possibly including some of the jobs with the same release time.

Before we present the proof for general m, we sketch the simplest case $m = 2$.

To prove that the algorithm is 1.8-competitive, we present a charging scheme. Consider a schedule A generated by the algorithm and an arbitrary (offline, adversarial) schedule Z. The charging scheme assigns each job in Z to some jobs in A. The assignment is allowed to be fractional, i.e., some fractions of a job in Z may be assigned to different jobs in A, assuming the total of these fractions

is 1. We then show that each job in A is charged at most 1.8, which completes the proof.

The charging scheme is defined as follows: Let t be the starting time of some job i in Z. If at time t, both machines in A are idle, then job i is charged 1 to itself in A. If at t one machine is busy in A, then i is charged 0.4 to the job running on that machine in A and 0.6 to itself in A. In both previous cases, charging to i is well-defined: Job i has to be scheduled in A, since there exists a feasible machine for i, namely the machine which is running no job at time t in the final schedule, and thus was also feasible for i at when BESTFIT decided i. Finally, if both machines are busy at t in A, then i is charged to the two jobs running in A at time t so that 0.4 is charged to the job that was decided first by the algorithm and 0.6 is charged to the other job. If job j is scheduled on a machine with the larger completion time at the time when BESTFIT decided j, then at any time when it is running in A j is the first job decided of the two currently running; thus j is charged at most $2 * 0.4 + 1 = 1.8$ from the two jobs started during its execution in Z and from itself. If job j is scheduled on the other machine in A, then the first machine is busy at all times when j can start in Z and thus j is charged at most $2 * 0.6 + 0.6 = 1.8$.

The instance showing that BESTFIT is no better than 1.8-competitive for $m = 2$ and $p \geq 3$ consists of these jobs: 3 jobs $(0, 6p+2)$, scheduled by BESTFIT on the first machine from time 0 to $3p$; 2 jobs $(1, 3p+2)$, scheduled on the second machine from time 1 to $2p + 1$; and finally 4 jobs $(2, 2p + 2)$ that are rejected. The optimum schedules all 9 jobs (essentially in the reverse order).

Now we are ready to present the full proof for a general m.

Theorem 1. *The competitive ratio of the algorithm* BESTFIT *is*

$$R_m = \frac{1}{1 - \left(\frac{m}{m+1}\right)^m}.$$

For $m \to \infty$, R_m decreases to $e/(e-1) \approx 1.582$; $R_2 = 1.8$, and $R_3 = 64/37 \approx 1.730$.

Proof. Let

$$Y_k = (m+1)^{k-1} m^{m-k} \quad \text{and} \quad X_k = \frac{Y_k}{(m+1)^m - m^m}.$$

Note that Y_k and X_k both increase with k and $X_1 + \cdots + X_m = 1$.

The upper bound is proved by a charging scheme. Consider a schedule A generated by the algorithm and an arbitrary schedule Z. Let i be a job in schedule Z and t its starting time in Z. Let $j_1, \ldots, j_k$ be all the jobs running (or just started) at time t in the schedule A, in the order they were decided by BESTFIT. We charge the total of 1 of job i in Z to jobs $j_1, \ldots, j_k$, and i in the schedule A so that we charge the amount of X_α to each j_α, $\alpha = 1, \ldots, k$ and the amount of $X_{k+1} + \cdots + X_m$ to i. Charging to i is well-defined: If $k < m$ then i is scheduled in A, as there exists a feasible machine for it at the time when it is considered,

namely the machine which is running no job at time t in A. Otherwise, if $k = m$, the amount to be charged to i is 0. We observe that the total charged is always equal to $X_1 + \cdots + X_m = 1$, and thus the definition of the charging scheme is sound.

To prove that the algorithm is R_m-competitive, it remains to show that each job in A is charged at most the amount of R_m; then the competitive ratio R_m follows by summing over all jobs. Suppose that some job j is scheduled on a machine with the kth largest $C(M_i)$ (since the first $k-1$ machines are not feasible). Then at any time from r_j up to (and including) $d_j - p$ at least $k-1$ machines are executing jobs decided before j. Thus j may be charged at most $X_k + \cdots + X_m$ from itself. Furthermore, at any time j is running in A, it is at most kth job decided by BestFit. It follows that the charge to j from each of the at most m other jobs started in Z while A is running j is at most X_i for some $i \leq k$, and this is at most X_k due to monotonicity of X_i's. The total charge to j is at most $mX_k + X_k + \cdots + X_m$. We claim that $mX_k + X_k + \cdots + X_m = R_m$, for any k: We have $mX_k + X_k = mX_{k+1}$ for any $k < m$, thus all the values are equal, and the last value is $mX_m + X_m = R_m$. This completes the proof of the desired competitive ratio.

An instance showing that BestFit is no better than R_m-competitive is this: Let $p > m$. First we present Y_m jobs $(0, 2Y_m p+m)$. Then, for each $k = m-1, m-2, \ldots, 1$, we have Y_k jobs $(m-k, Y_{k+1}p + m)$. Finally, mY_1 jobs $(m, Y_1 p + m)$ arrive.

It can be verified that the algorithm first schedules all Y_m jobs with $r_j = 0$ on one machine, then all Y_{m-1} jobs with $r_j = 1$ on the next machine, and so on, and it rejects the mY_1 jobs with $r_j = m$. On the other hand, the optimal solution is idle until time m, then schedules the mY_1 jobs with $r_j = m$, then schedules the Y_1 jobs with $r_j = m - 1$, and so on, completing all the jobs. The ratio is equal to R_m. □

3 The Lower Bounds for Immediate Decision

If the instance starts by a modest number of jobs with a very large deadline, BestFit schedules them on one machine close to time 0. Apparently this gives a big advantage to the adversary, who can then focus on the region where these jobs are scheduled. It would seem reasonable to try to improve the performance by spreading these jobs somehow uniformly over the whole feasible time interval. However, perhaps surprisingly, for $m = 2$, we can prove that no such strategy helps and in fact BestFit is an optimal algorithm with immediate decision.

In the lower bound proof for $m = 2$, we try to force the algorithm to schedule the first jobs so that both processors are busy at some time steps. Then we release pairs of tight jobs that must be rejected. Typically, we create two such problematic times after scheduling of 5 jobs. This results in two additional pairs, i.e., a total of 9 jobs of which the algorithm schedules only 5. The optimum always schedules all the jobs. The first jobs have a very long feasible intervals, so in an optimal schedule they can always be moved so that they do not conflict with

any other jobs. The optimal schedule of tight jobs is determined, so it remains to verify in each case that the remaining jobs can be scheduled without conflicts with the tight jobs. The number of cases is relatively high also due to the fact that some of the jobs may be rejected.

Theorem 2. *Let A be a deterministic algorithm with immediate decision for $m = 2$ machines and $p \geq 4$. Then A is no better than* 1.8*-competitive.*

Proof. We start by releasing three jobs $(0, 100p)$. Now we wait for the algorithm to decide the schedule of these jobs. Note that this means that time advances to 1 and we cannot release more jobs with $r_j = 0$. We note that optimal schedule can always schedule these jobs so that they do not overlap with a feasible time interval of any other job. Thus in the rest of the proof it is sufficient to verify that the other jobs can be scheduled. First we analyze the case of all three jobs accepted. We renumber them so that their start times are $S_1 \leq S_2 \leq S_3$.

1. There exist two times $t_2 > t_1 \geq p$, where two jobs are running. This includes the case of two jobs overlapping for more than one time step.
   [In this case the algorithm essentially gives up.]
   We release two tight jobs $(t_1 - p + 1, t_1 + 1)$ and two tight jobs $(t_2, t_2 + p)$. This is possible since $t_1 - p + 1 \geq 1$. The algorithm has to reject all four new jobs. The optimal schedule schedules all seven jobs, since $t_2 \geq t_1 + 1$. Thus the competitive ratio is no better than $7/3$.
2. We have $S_3 - S_2 \leq 2p - 1$ and $S_2 \geq p$. Since the previous case does not hold, we also have $S_3 - S_2 \geq p - 1$.
   [This case matches the behavior of BestFit.]
   We release two jobs, 4 and 5, with $(r_j, d_j) = (S_3 - p - 2, S_3 + 2p - 1)$. This is possible as $S_3 - p - 2 \geq S_2 - 3 \geq 1$.
   If any of jobs 4 and 5 gets scheduled, then it overlaps the schedule of jobs 2 or 3.
   Depending on the schedule of jobs 4 and 5, we schedule two pairs of tight jobs, choosing from pairs with (r_j, d_j) equal to $(S_3 - p - 1, S_3 - 1)$, $(S_3 - 1, S_3 + p - 1)$, or $(S_3 + p - 1, S_3 + 2p - 1)$. We choose the pairs as follows.
   (a) If the algorithm scheduled both jobs 4 and 5, it can be verified that two of the slots for the tight jobs contain a time when both machines are busy. Release these two pairs.
   (b) If the algorithm schedules only one of jobs 4 and 5, there will be one slot for the tight jobs containing a time when both machines are busy. Release this pair and an arbitrary additional pair.
   (c) If the algorithm rejects both jobs 4 and 5, release arbitrary two pairs.
   In every case it can be checked that the algorithm schedules at most 5 of the total 9 jobs. Also, the optimum schedules all jobs, as 4 and 5 can be scheduled in the unused slot for tight jobs.
3. We have $S_3 - S_2 \geq 2p$ and $S_2 \geq p$.
   [This case is most interesting, as it kills attempts at algorithms that try to spread the jobs with long feasible intervals.]

We release job 4 with $(r_4, d_4) = (S_2 - 2, S_2 + 2p - 1)$. Depending on its schedule, we release two tight jobs 5 and 6 as follows: if $S_4 \leq S_2 - 1$, then they are $(S_2 - 1, S_2 + p - 1)$, otherwise $(S_2 + p - 1, S_2 + 2p - 1)$ (including the case when job 4 is rejected). The algorithm can schedule only one job of 4, 5, and 6.

We continue similarly by job 7 with $(r_7, d_7) = (S_3 - 2, S_3 + 2p - 1)$. Depending on its schedule, we release two tight jobs 8 and 9 as follows: if $S_7 \leq S_3 - 1$, then they are $(S_3 - 1, S_3 + p - 1)$, otherwise $(S_3 + p - 1, S_3 + 2p - 1)$ (including the case when job 7 is rejected). The algorithm can schedule only one job of 7, 8, and 9.

Note that $r_7 \geq d_4 - 1$. This guarantees that the optimum can schedule both jobs 4 and 7, no matter which tight jobs are released.

Overall, the competitive ratio is at least 9/5.

4. It remains to handle the case when $S_2 < p$.
[It is not very smart if the algorithm overlaps the jobs at the very beginning, but it seems that unfortunately we need to handle this case separately, revisiting some of the previous cases.]

We release two tight jobs 4 and 5 with a slot $(1, p+1)$, which get rejected, and two jobs 6 and 7 with $(r_j, d_j) = (1, 100p)$. Out of the four jobs 1, 2, 4, and 5, the algorithm schedules only two.

If there is a time $t \geq p + 1$ when both machines are busy, we conclude the proof by another pair of tight jobs. Similarly, if one of the jobs 6 and 7 is rejected, release a pair of tight jobs with $r_j \geq p + 1$ overlapping job 3.

Otherwise, renumber jobs 3, 6, and 7 to 1, 2, and 3, subtract t from all times, and iterate the case of three released and scheduled jobs once more. Since the three jobs do not overlap, we end up in case 2 or 3. It can be easily verified that the optimum can schedule all the jobs from the first and second iterations together. The ratio is at least 9/5 for the second iteration, so including the 4 jobs in the first iteration, the overall ratio is strictly worse than 9/5.

Now it remains to analyze the case that algorithm rejected some of the first three jobs. If the algorithm rejects at least two jobs then the ratio is at least 3.

If the algorithm rejects one job, renumber the jobs so that $S_1 \leq S_2$ and 3 is rejected. If $S_2 < 3$, release two tight jobs $(1, p + 1)$, and the ratio is 5/2. If $S_2 \geq 3$, we release job 4 with $(r_4, d_4) = (S_2 - 2, S_2 + 2p - 1)$. Depending on its schedule, we release two tight jobs 5 and 6 as follows: if $S_4 \leq S_2 - 1$, then they are $(S_2 - 1, S_2 + p - 1)$, otherwise $(S_2 + p - 1, S_2 + 2p - 1)$ (including the case when job 4 is rejected). The algorithm can schedule only one job of 4, 5, and 6. Thus it schedules only 3 jobs, while the optimum schedules all 6. □

For an arbitrary m, we prove a lower bound of 4/3 even for randomized algorithms. This is an easy adaptation of the standard bounds for a single machine.

Theorem 3. *Let A be an algorithm (possibly randomized) with immediate decision for m machines and $p \geq 2$. Then A is no better than 4/3-competitive.*

Proof. Release m jobs $(0, 2p+1)$. If the expected number of jobs that start at time 0 or 1 is at least $m/2$, release m jobs $(1, p+1)$. Otherwise, release m jobs $(p, 2p)$. In both cases, the algorithm schedules at most $3m/2$ jobs, while the optimum is $2m$. □

4 Jobs with a Small Length

The previous lower bounds hold only for sufficiently large p. It is an interesting question what happens for smaller values of p, in particular for unit jobs, i.e., $p = 1$. Note that without the requirement of immediate decision, for $p = 1$, it is easy to generate an optimal schedule online for any number of machines: Always schedule the m jobs with the smallest deadlines among those which are still feasible.

With immediate decision for $m = 2$, we prove that deterministic algorithms are at least 1.6-competitive for $p = 1$ and 5/3-competitive for $p = 2$. Finally, for $p = 3$, we revisit the proof of Theorem 2 and show that by handling one case more carefully we obtain the same lower bound of 1.8 as for $p \geq 4$.

For the case of unit jobs, we need an auxiliary lemma.

Lemma 1. *Suppose we have a discrete interval $I = [1..N]$, non-negative integral weights w_i, $i \in I$, such that $\sum_{i \in I} w_i = N/2$, and an arbitrary number $k \leq n$. Then there exist three disjoint subintervals I_1, I_2, I_3 of I, each with total weight $\sum_{i \in I_\alpha} w_i \geq |I_\alpha|/2$, with total length $|I_1| + |I_2| + |I_3| = k$, such that in addition I_1 contains 1 and I_3 contains N.*

Proof. If there exists an interval I_2 of length k with weight at least $k/2$, we set $I_1 = I_3 = \emptyset$ and we are done. Otherwise by averaging there exists two intervals $[1..i]$ and $[i'..N]$, with total length k and total weight at least $k/2$. Thus one of these intervals has weight at least half of its length. W.l.o.g. assume that the interval $[1..i]$ has weight at least $i/2$. Since the interval $[1..k]$ has weight less than $k/2$, it follows that for some j, $i \leq j < k$, the interval $[1..j]$ has weight exactly $j/2$. Then we apply the lemma recursively on the interval $I - [1..j]$ and with $k - j$ in place of k. We merge the interval $[1..j]$ with I_1 resulting from the recursive application. □

Theorem 4. *Let A be a deterministic algorithm with immediate decision for $m = 2$ and $p = 1$. Then A is no better than 1.6-competitive.*

Proof. Let M be large. In the first phase, we release $2M$ jobs $(0, 4M)$. If some job is not scheduled, we assign it to any time arbitrarily but so that at no time two jobs are scheduled or assigned.

We apply Lemma 1 to the interval $[0..4M-1]$ and $k = 3M$ with the weights equal to the number of jobs scheduled at or assigned to the given time. It follows that there exist three intervals $[t_i, t_i')$, $i = 1, 2, 3$, such that denoting $T_i = t_i' - t_i$ their lengths, their total length is $T_1 + T_2 + T_3 = 3M$ and each interval contains at least $T_i/2$ jobs. Denote n_i the number of jobs scheduled in $[t_i, t_i')$ or assigned to a time in $[t_i, t_i')$.

Next, in the second phase, for each interval $i = 1, 2, 3$, we release $\max\{3T_i/2 - n_i - 2, 0\}$ jobs $(t_i + 1, t'_i)$. Note that there are at most T_i new jobs. Assign all unscheduled jobs and reassign all the jobs from the first phase assigned originally to this interval to times in this interval arbitrarily but so that at no time two jobs are scheduled or assigned. (It may be necessary to reassign some job from the first phase, since the algorithm schedules a job to the same time).

As there are total $3T_i/2-2$ jobs in the interval $[t_i, t'_i)$ from the first two phases, there are at least $T_i/2 - 2$ times when two jobs are scheduled or assigned. For exactly $T_i/2 - 4$ of such times $t > t_i + 1$ release two tight jobs $(t, t + 1)$ in the third phase. (We have to omit the first two times in each interval, since after observing the assignment of the jobs from the second phase released at $t_i + 1$ we cannot use them).

The optimum can schedule all the jobs: The tight jobs take less than $T_i/2$ times in each interval, and the at most T_i jobs from the second phase can be scheduled in the remaining available $T_i/2$ times in the interval. The jobs from the first phase exactly fit outside the three chosen intervals.

In the first two phases, there are at most $2M + T_1 + T_2 + T_3 = 5M$ jobs which may be scheduled by the algorithm. In the third phase, there are exactly $2(T_1/2 + T_2/2 + T_3/2) - 24 = 3M - 24$ jobs, all rejected by the algorithm. Thus the competitive ratio is at least $(8M - 24)/5M$, which is arbitrarily close to 1.6 for large M. □

Theorem 5. *Let A be a deterministic algorithm with immediate decision for $m = 2$ machines and $p = 2$. Then A is no better than 5/3-competitive.*

Proof. First, we release M jobs $(0, 100M)$, for a large M. Next we search the current schedule from beginning to end for times where both machines are busy. We start at time 1, as we are not allowed to release other jobs with $r_j = 0$. Whenever we find such time t, we release two jobs $(t, t + 2)$. We group them with the two jobs scheduled at t and also with the last job scheduled before t, if it is not included in the previous group. So we have groups of at most 5 jobs, such that the algorithm rejects two jobs in each group. We continue the search at time $t + 2$.

Now no two remaining jobs (i.e., those not in any group) overlap in the schedule. Moreover for each remaining job j, we released no tight job overlapping with the interval $[S_j, S_j + 4)$.

Next, we examine the remaining jobs again from beginning of the schedule to end. We skip the first job as it may be scheduled at time 0 or 1 (this is why we need M large). Suppose that we are examining job j.

If there is no remaining job scheduled in time $[S_j, S_j + 4)$, we release two jobs $(S_j - 1, S_j + 4)$. Then, if one of them is scheduled and starts at time $S_j + 1$ or earlier, we release two tight jobs $(S_j, S_j + 2)$; otherwise we release two tight jobs $(S_j + p, S_j + p + 2)$. We create a group of j and these four jobs. The algorithm schedules at most 3 out of this group of 5 jobs.

Otherwise, there is some job j' with $S_{j'} \leq S_j + 3$. We release only one job (S_j-1, S_j+4). Again we continue with two tight jobs (S_j, S_j+2) or (S_j+2, S_j+4), so that the algorithm can schedule only one of the three new jobs. Again j, j'

and these three jobs form a new group. We skip j' and continue examining the following jobs.

Now we have all jobs but the first one in groups of 5. Each group has at most 3 jobs scheduled by A. In an optimal schedule, all tight jobs can be scheduled, then all jobs with $d_j = r_j + 5$ can be scheduled with $S_j = r_j + 1$ or $S_j = r_j + 3$. Finally, there is plenty of room for the remaining jobs (released at time 0). Thus any optimal schedule schedules all the jobs.

For M large enough, this proves that no algorithm is $(5/3 - \varepsilon)$-competitive for $\varepsilon > 0$. □

Theorem 6. *Let A be a deterministic algorithm with immediate decision for $m = 2$ machines and $p \geq 3$. Then A is no better than 1.8-competitive.*

Proof. We need to revisit the proof for $p \geq 4$. All cases but 2 can be analyzed exactly same way. Case 2 must be handled more carefully. If the algorithm does not reject any job among 1, 2, 3, and $S_2 = 3$, $S_3 = 5$ we cannot follow the case 2 as we would need to release a job with $r_4 = 0$. In this case we release six jobs $(1, 10)$. Algorithm can accept only two of them, so overall it schedules only 5 out of 9 jobs. □

5 Conclusions, Open Problems, Acknowledgments

The main open problem in this area remains to design optimal or at least good algorithms for the unrestricted model and $m > 2$. Despite the good progress we have been able to achieve using algorithms with immediate decision, one would expect that also for $m > 2$, the best algorithms will use the flexibility of the unrestricted model. However, no such algorithms are known.

We know that for $m = 2$, immediate decision increases the optimal competitive ratio and our new algorithm is optimal in the restricted model. For $m \geq 3$ we would expect the same to be true, but we have no lower bounds. It would be also interesting to prove more in the case of unit jobs, either for $m = 2$ or for larger m.

Finally, virtually nothing is known about randomized algorithms for $m \geq 2$.

We are grateful to anonymous referees for many comments that helped us to improve the presentation of this paper. T. Ebenlendr and J. Sgall were partially supported by Institutional Research Plan No. AV0Z10190503, by Inst. for Theor. Comp. Sci., Prague (project 1M0545 of MŠMT ČR), and grant 201/05/0124 of GA ČR. G. Zhang was partially supported by NSFC (60573020).

References

1. Baptiste, P., Brucker, P., Knust, S., Timkovsky, V.: Ten notes on equal-execution-time scheduling. 4OR 2, 111–127 (2004)
2. Baruah, S.K., Haritsa, J., Sharma, N.: On-line scheduling to maximize task completions. J. Comb. Math. Comb. Comput. 39, 65–78 (2001) (A preliminary version appeared. Proc. 15th Real-Time Systems Symp., IEEE, pp. 228–236 (1994))

3. Chrobak, M., Jawor, W., Sgall, J., Tichý, T.: Online scheduling of equal-length jobs: Randomization and restarts help. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 358–370. Springer, Heidelberg (2004)
4. Ding, J., Zhang, G.: Online scheduling with hard deadlines on parallel machines. In: Cheng, S.-W., Poon, C.K. (eds.) AAIM 2006. LNCS, vol. 4041, pp. 32–42. Springer, Heidelberg (2006)
5. Goldman, S.A., Parwatikar, J., Suri, S.: Online scheduling with hard deadlines. J. Algorithms 34, 370–389 (2000)
6. Goldwasser, M.H., Kerbikov, B.: Admission control with immediate notification. J. Sched. 6, 269–285 (2003)
7. Goldwasser, M.H., Pedigo, M.: Online, non-preemptive scheduling of equal-length jobs on two identical machines. In: Arge, L., Freivalds, R. (eds.) SWAT 2006. LNCS, vol. 4059, pp. 113–123. Springer, Heidelberg (2006)

k-Anonymization with Minimal Loss of Information

Aristides Gionis[1] and Tamir Tassa[2]

[1] Yahoo! Research, Barcelona, Spain
gionis@yahoo-inc.com
[2] Division of Computer Science, The Open University, Ra'anana, Israel
tamirta@openu.ac.il

Abstract. The technique of k-anonymization allows the releasing of databases that contain personal information while ensuring some degree of individual privacy. Anonymization is usually performed by generalizing database entries. We formally study the concept of generalization, and propose two information-theoretic measures for capturing the amount of information that is lost during the anonymization process. Those measures are more general and more accurate than those proposed in [19] and [1]. We study the problem of achieving k-anonymity with minimal loss of information. We prove that it is NP-hard and study polynomial approximations for the optimal solution. Our first algorithm gives an approximation guarantee of $O(\ln k)$ – an improvement over the best-known $O(k)$-approximation of [1]. As the running time of the algorithm is $O(n^{2k})$, we also show how to adapt the algorithm of [1] in order to obtain an $O(k)$-approximation algorithm that is polynomial in both n and k.

1 Introduction

Consider a database that holds information on individuals in some population $U = \{u_1, \ldots, u_n\}$. Each individual is described by a collection of r public attributes (also known as *quasi-identifiers*), $A_1, \ldots, A_r$, and s private attributes, $Z_1, \ldots, Z_s$. Each of the attributes consists of several possible values: $A_j = \{a_{j,\ell} : 1 \le \ell \le m_j\}$, $1 \le j \le r$, and $Z_j = \{z_{j,\ell} : 1 \le \ell \le n_j\}$, $1 \le j \le s$. For example, if A_j is gender then $A_j = \{M, F\}$, while if it is the age of the individual, it is a bounded nonnegative natural number. The public database holds all publicly available information on the individuals in U; it takes the form,

$$D = \{R_1, \ldots, R_n\}\,, \quad \text{where } R_i \in A_1 \times \cdots \times A_r\,, \quad 1 \le i \le n\,.$$

The corresponding private database holds the private information,

$$D' = \{S_1, \ldots, S_n\}\,, \quad \text{where } S_i \in Z_1 \times \cdots \times Z_s\,, \quad 1 \le i \le n\,.$$

The complete database is the concatenation of those two databases, $D \| D' = \{R_1 \| S_1, \ldots, R_n \| S_n\}$. We refer hereinafter to the tuples R_i and S_i, $1 \le i \le n$, as

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 439–450, 2007.

(public or private) records. The j-th component of the record R_i (namely, the (i,j)-th entry in the database D) will be denoted hereinafter by $R_i(j)$.

Such databases may be of interest to the general public even though they hold information on individuals. The goal is to reveal information in order to allow data mining, while respecting the privacy of the individuals that are represented in the database. Many approaches were suggested for playing this delicate game that requires finding the right path between data hiding and data disclosure. Such approaches include query auditing [10,16,17], output perturbation [6,10,11], secure multi-party computation [2,13,14,18,24], and data sanitization [3,4,5,7,12]. One of the recent approaches, proposed by Samarati and Sweeney [20,21,22] is k-anonymization. The main idea in this approach is to suppress or generalize some of the public data in the database so that each of the public records becomes indistinguishable from at least $k-1$ additional records. Consequently, the private data may be linked to sets of individuals of size no less than k, whence the privacy of the individuals is protected to some extent.

The problem that we study here is the problem of k-anonymization with minimal loss of information: Given a public database D, and acceptable generalization rules for each of its attributes, find its "nearest" k-anonymization; namely, find a k-anonymization of D that conceals a minimum amount of information. Meyerson and Williams [19] introduced this problem and studied it under the assumption that database entries may be either left intact or totally suppressed. In that setting, the goal is to achieve k-anonymity while minimizing the number of suppressed entries. They showed that the problem is NP-hard and devised two approximation algorithms for that problem: One that runs in time $O(n^{2k})$ and achieves an approximation ratio of $O(k \ln k)$; and another that has a fully polynomial running time (namely, it depends polynomially on both n and k) and guarantees an approximation ratio of $O(k \ln n)$. Aggarwal et al. [1] extended the setting of suppressions-only by allowing more general rules for generalizing database entries towards achieving k-anonymity. They proposed a way of penalizing each such action of generalizing a database entry and showed that the problem of achieving k-anonymity in that setting with minimal penalty is NP-hard. They then devised an approximation algorithm for that problem that guarantees an approximation ratio of $O(k)$.

In this study we extend the framework of k-anonymization to include any type of generalization operators and define two measures of loss of information that are both more general and more accurate than the measure that was used in [1] (the measure that was used in [19] is a special case of the one that was used in [1]). We call these measures *the entropy measure* and *the monotone entropy measure*. We show that the problem of k-anonymization with minimal loss of data (measured by either of those measures) is NP-hard. We then proceed to describe an approximation algorithm with an approximation guarantee of $O(\ln k)$—a significant improvement over the previous best result of $O(k)$. The algorithm applies to both of our measures, as well as the measures that were used in [19] and [1]. We note that Meyerson and Williams [19] hypothesized that k-anonymization cannot be approximated, in polynomial time, with an

approximation factor that is $o(\ln k)$. What enabled this significant improvement was our novel approach to this approximation problem. The approximation algorithms in both [19] and [1] were based on the so-called *graph representation*. In [1] it was shown that using the graph representation it is impossible to achieve an approximation ratio that is better than $\Theta(k)$. We were able to offer the significantly better $O(\ln k)$ approximation ratio by breaking out of the graph representation framework and using a *hypergraph* approach instead.

The paper is organized as follows: In Section 2 we give a precise definition of what is generalization, and we describe and illustrate several natural types of generalization. In Section 3 we define and discuss our two measures of loss of information. In Section 4 we define the problem of k-anonymization with minimal loss of information and state its NP-hardness with respect to both measures of loss of information. In Section 5 we present an algorithm that approximates optimal k-anonymity with approximation ratio of $O(\ln k)$, for the entropy and monotone entropy measures. The running time of that algorithm is $O(n^{2k})$. We then proceed to describe how to adapt the approximation algorithm of [1] to achieve an $O(k)$-approximation ratio with respect to our measures, in time that is polynomial in both n and k.

Due to lack of space, all proofs are omitted from this version and may be found in the full version of this paper.

2 Generalization

The basic technique for obtaining k-anonymization is by means of *generalization*. By generalization we refer to the act of replacing the values that appear in the database with subsets of values, so that entry $R_i(j)$, $1 \leq i \leq n$, $1 \leq j \leq r$, which is an element of A_j, is replaced by a subset of A_j that includes that element.

Definition 1. *Let A_j, $1 \leq j \leq r$, be finite sets and let $\overline{A}_j \subseteq \mathcal{P}(A_j)$ be a collection of subsets of A_j. A mapping $g : A_1 \times \cdots \times A_r \to \overline{A}_1 \times \cdots \times \overline{A}_r$ is called a generalization if for every $(b_1, \ldots, b_r) \in A_1 \times \cdots \times A_r$ and $(B_1, \ldots, B_r) = g(b_1, \ldots, b_r)$, it holds that $b_j \in B_j$, $1 \leq j \leq r$.*

We illustrate the concept of generalization by several examples of natural generalization operators.

Generalization by suppression. Assume that $\overline{A}_j = A_j \cup \{A_j\}$ for all $1 \leq j \leq r$ and that g either leaves entries unchanged or replaces them by the entire set of attribute values, i.e., $g(b_1, \ldots, b_r) = (\overline{b}_1, \ldots, \overline{b}_r)$, where $\overline{b}_j \in \{b_j, *\}$, and $*$ denotes an element outside $\bigcup_{1 \leq j \leq r} A_j$. In that case we refer to g as *generalization by suppression*.

Generalization by hierarchical clustering trees. Aggarwal et al. [1] considered a setting in which for every attribute A_j there is a corresponding balanced tree, $\mathcal{T}(A_j)$, that describes a hierarchical clustering of A_j. Each node of $\mathcal{T}(A_j)$ represents a subset of A_j, the root of the tree is the entire set A_j, the descendants of each node represent a partition of the subset that corresponds to the ancestor

node, and the leaves correspond to the singleton subsets. Given such a balanced tree, they considered generalization operators that may replace an entry $R_i(j)$ with any of the ancestors of $R_i(j)$ in $\mathcal{T}(A_j)$. Generalization by suppression is a special case of generalization by clustering trees where all trees are of height 2.

Some of our results require that the collection of subsets $\overline{A}_j$, $1 \leq j \leq r$, satisfy the following natural property.

Definition 2. *Given an attribute $A = \{a_1, \ldots, a_m\}$, a corresponding collection of subsets $\overline{A}$ is called proper if (i) it includes all singleton subsets $\{a_i\}$, $1 \leq i \leq m$, (ii) it includes the entire set A, and (iii) it is a laminar collection in the sense that $B_1 \cap B_2 \in \{\emptyset, B_1, B_2\}$ for all $B_1, B_2 \in \overline{A}$.*

So far we spoke of generalizations of records. We now turn to speak of generalizations of an entire database.

Definition 3. *Let $D = \{R_1, \ldots, R_n\}$ be a database having public attributes $A_1, \ldots, A_r$, let $\overline{A}_1, \ldots, \overline{A}_r$ be corresponding collections of subsets, and let $g_i : A_1 \times \cdots \times A_r \to \overline{A}_1 \times \cdots \times \overline{A}_r$ be generalization operators. Denoting $\overline{R}_i := g_i(R_i)$, $1 \leq i \leq n$, the database $g(D) := \{\overline{R}_1, \ldots, \overline{R}_n\}$ is called a generalization of D.*

We conclude this section with the following definition of a partial order on the set of generalized records:

Definition 4. *Define a relation $\sqsubseteq$ on $\overline{A}_1 \times \cdots \times \overline{A}_r$ as follows: If $R, R' \in \overline{A}_1 \times \cdots \times \overline{A}_r$ then $R \sqsubseteq R'$ if and only if $R(j) \subseteq R'(j)$ for all $1 \leq j \leq r$.*

3 Measures of Loss of Information

3.1 Previously Used Measures

The quality of a k-anonymization of a given database is typically measured by the amount of information that is lost due to generalization. Meyerson and Williams [19] concentrated on the case of generalization by suppression. Their measure of loss of information was the number of generalized entries (namely, *s) in the k-anonymized database. Aggarwal et al. [1] considered generalizations by hierarchical clustering trees and proposed to penalize by r/ℓ_j each generalization of an entry $R_i(j)$ to a subset residing at the r-th level of the hierarchical clustering tree $\mathcal{T}(A_j)$, the height of which is ℓ_j. The tree measure is a generalization of the measure proposed by Meyerson and Williams.

We find the tree measure quite arbitrary. For example, if one attribute is gender and another attribute is age, the loss of information by concealing the gender is much less than that incurred by concealing the age. Also, the levels of the trees $\mathcal{T}(A_j)$ need not be equally-spaced in terms of information loss.

3.2 The Entropy Measure

Following [9] and [23], we suggest to use the standard measure of information, namely entropy, in order to assess more accurately the amount of information that is lost by anonymization.

The public database $D = \{R_1, \ldots, R_n\}$ induces a probability distribution for each of the public attributes. Let X_j, $1 \leq j \leq r$, denote hereinafter the value of the attribute A_j in a randomly selected record from D. Then

$$\Pr(X_j = a) = \frac{\#\{1 \leq i \leq n : R_i(j) = a\}}{n}\,.$$

Let B_j be a subset of A_j. Then the conditional entropy $H(X_j|B_j)$ is defined as

$$H(X_j|B_j) = -\sum_{b \in B_j} \Pr(X_j = b|X_j \in B_j) \log_2 \Pr(X_j = b|X_j \in B_j)\,.$$

Note that if $B_j = A_j$ then $H(X_j|B_j) = H(X_j)$ while in the other extreme case where B_j consists of one element, we have zero uncertainty, $H(X_j|B_j) = 0$. This allows us to define the following cost function of a generalization operator:

Definition 5. *Let $D = \{R_1, \ldots, R_n\}$ be a database having public attributes $A_1, \ldots, A_r$, and let X_j be the random variable that equals the value of the j-th attribute A_j, $1 \leq j \leq r$, in a randomly selected record from D. Then if $g(D) = \{\overline{R}_1, \ldots, \overline{R}_n\}$ is a generalization of D,*

$$\Pi_e(D, g(D)) = \sum_{i=1}^{n} \sum_{j=1}^{r} H(X_j|\overline{R}_i(j)) \tag{1}$$

is the entropy measure of the loss of information caused by generalizing D into $g(D)$.

The Non-monotonicity of the Entropy Measure. A natural property that one might expect from any measure of loss of information is monotonicity:

Definition 6. *Let $D = \{R_1, \ldots, R_n\}$ be a database and let Π be any measure of loss of information. Then Π is called monotone if $\Pi(D, g(D)) \leq \Pi(D, g'(D))$ for any two D-generalizations, $g(D)$ and $g'(D)$, where $g(D)_i \sqsubseteq g'(D)_i$ for all $1 \leq i \leq n$.*

The tree measure is clearly monotone. The entropy measure Π_{e}, on the other hand, is not always monotone. The non-monotonicity of the entropy measure may always be rectified, in the sense that for any collection of subsets of a given attribute, $\overline{A}$, it is always possible to find a partial collection, $\hat{A} \subseteq \overline{A}$, so that the entropy measure is monotone on $\hat{A}$. Due to lack of space, we postpone the discussion of the non-monotonicity of the entropy measure to the full version of this paper. There we exemplify it, discuss it, and show how to rectify it.

3.3 The Monotone Entropy Measure

Here we introduce the *monotone entropy measure*, a simple variant of the entropy measure that respects monotonicity.

Definition 7. *Let $D = \{R_1, \ldots, R_n\}$ be a database having public attributes $A_1, \ldots, A_r$, and let X_j be the random variable that equals the value of the j-th attribute A_j, $1 \leq j \leq r$, in a randomly selected record from D. Then if $g(D) = \{\overline{R}_1, \ldots, \overline{R}_n\}$ is a generalization of D,*

$$\Pi me(D, g(D)) = \sum_{i=1}^{n} \sum_{j=1}^{r} \Pr(\overline{R}_i(j)) \cdot H(X_j | \overline{R}_i(j)) \tag{2}$$

is the monotone entropy measure of the loss of information caused by generalizing D into $g(D)$.

Comparing (2) to (1), we see that each of the conditional entropies is multiplied by the corresponding probability. The monotone entropy measure coincides with the entropy measure when considering generalization by suppressions only. However, when the collections of subsets $\overline{A}_j$ include also intermediate subsets, the entropy that is associated with such a subset is multiplied by the probability of the subset. Since this multiplier increases as the subset includes more elements, the monotone entropy measure penalizes generalizations more than the entropy measure does.

Lemma 1. *The monotone entropy measure is monotone.*

4 *k*-Anonymization with Minimal Loss of Data

We are now ready to define the concepts of k-anonymization and the corresponding problem of k-anonymization with minimal loss of information.

Definition 8. *A k-anonymization of a database $D = \{R_1, \ldots, R_n\}$ is a generalization $g(D) = \{\overline{R}_1, \ldots, \overline{R}_n\}$ where for all $1 \leq i \leq n$ there exist indices $1 \leq i_1 < i_2 < \cdots < i_{k-1} \leq n$, all of which are different from i, such that $\overline{R}_i = \overline{R}_{i_1} = \cdots = \overline{R}_{i_{k-1}}$.*

k-ANONYMIZATION: Let $D = \{R_1, \ldots, R_n\}$ be a database having public attributes A_j, $1 \leq j \leq r$. Given collections of attribute values, $\overline{A}_j \subseteq \mathcal{P}(A_j)$, $1 \leq j \leq r$, and a measure of information loss Π, find a k-anonymization $g(D) = \{\overline{R}_1, \ldots, \overline{R}_n\}$, where $\overline{R}_i \in \overline{A}_1 \times \cdots \times \overline{A}_r$, $1 \leq i \leq n$, that minimizes $\Pi(D, g(D))$.

The following theorem is an adaptation of [19, Theorem 3.1].

Theorem 1. *The problem of k-ANONYMIZATION with generalization by suppression, where the measure of loss of information is the entropy measure (1), $\Pi = \Pi e$, or the monotone entropy measure (2), $\Pi = \Pi me$, is NP-hard for $k \geq 3$, if $|A_j| \geq k + 1$ for all $1 \leq j \leq r$.*

5 Approximating Optimal *k*-Anonymity

In this section we describe two approximation algorithms for the problem of k-anonymization with minimal loss of information. We assume here that all collections of subsets are proper. The first algorithm, described in Sections 5.1-5.3,

achieves an approximation ratio of $O(\ln k)$—a significant improvement with respect to the best known $O(k)$-approximation algorithm [1]. As that algorithm runs in time $O(n^{2k})$, we show in Section 5.4 that the $O(k)$-approximation algorithm of [1] that runs in time $O(kn^2)$ may be used also for approximating optimal k-anonymity when using the entropy and monotone entropy measures. The question of the existence of a fully polynomial approximation algorithm with an $o(k)$-approximation ratio remains open.

5.1 The Generalization Cost of Subsets

Any k-anonymization of D defines a clustering (namely, a partition) of D where each cluster consists of all records that were replaced by the same generalized record. In order to lose a minimal amount of information, all records in the same cluster are replaced with the minimal generalized record that generalizes all of them. To that end we define the closure of a set of records.[1]

Definition 9. *Let $A_1, \ldots, A_r$ be attributes with corresponding collections of subsets $\overline{A}_1, \ldots \overline{A}_r$ that are all proper. Then given $M \subseteq A_1 \times \cdots \times A_r$, its closure is defined as*

$$\overline{M} = \min_{\sqsubseteq} \left\{ C \in \overline{A}_1 \times \cdots \times \overline{A}_r : R \sqsubseteq C \text{ for all } R \in M \right\}.$$

Definition 10. *Let $D = \{R_1, \ldots, R_n\}$ be a database with attributes $A_1, \ldots, A_r$, having proper collections of subsets $\overline{A}_1, \ldots \overline{A}_r$. Let X_j be the value of the attribute A_j in a randomly selected record from D. Then given a subset of records, $M \subseteq D$, its generalization cost by the entropy measure is,*

$$d(M) = d_e(M) = \sum_{j=1}^{r} H(X_j | \overline{M}_j)\,, \tag{3}$$

while its generalization cost by the monotone entropy measure is,

$$d(M) = d_{me}(M) = \sum_{j=1}^{r} \Pr(\overline{M}_j) \cdot H(X_j | \overline{M}_j)\,. \tag{4}$$

The generalization cost of M is therefore the amount of information that we lose for each record $R \in M$ if we replace it by the minimal generalized record $\overline{M}$.

We noted earlier that the entropy measure is not necessarily monotone. However, as explained in the full version, this problem rarely occurs. In addition, as we show there, we may always avoid it by narrowing down the collections $\overline{A}_j$, $1 \leq j \leq r$, until the entropy measure becomes monotone with respect to them. For the sake of simplicity, we assume monotonicity hereinafter. Namely,

$$M \subseteq M' \subseteq A_1 \times \cdots \times A_r \text{ implies that } d(M) \leq d(M')\,. \tag{5}$$

[1] In our discussion, a *set* actually means a *multiset*; namely, it may include repeated elements.

If we use the generalization cost by the monotone entropy measure, $d(M) = d_{\mathrm{me}}(M)$, then (5) always holds.

The notion of the generalization cost of a set of records is related to the notion of the *diameter* of such a set, as defined in [19]. The diameter of a set of records $M \subseteq A_1 \times \cdots \times A_r$ was defined as

$$\mathrm{diam}(M) = \max_{R,R' \in M} \delta(R, R'), \quad \delta(R, R') := |\{1 \leq j \leq r : R(j) \neq R'(j)\}| \,. \tag{6}$$

In other words, if the two records R and R' were to be generalized by means of suppression, $\mathrm{dist}(R, R')$ equals the minimal number of attributes that would be suppressed in each of the two records in order to make them identical.

Our notions of generalization cost, (3) and (4), and the notion of the diameter, (6), are functions that associate a *size* to a given set of records. Our notions, though, of generalization cost, improve that of the diameter as follows:

1. The generalization costs, (3) and (4), generalize the definition of the diameter, (6), in the sense that they apply to any type of generalization (the definition of the diameter is restricted to generalization by suppression).
2. The notions of the generalization cost use the more accurate entropy and monotone entropy measures (the definition of the diameter only counts the number of suppressed entries).
3. Most importantly, while the size of a set of records that is defined in (6) is a *diameter* (namely, it is based on pairwise distances), the size that is defined in (3) and (4) is a *volume*. All three notions offer measures for the amount of information that is lost if the entire set of records, M, is to be anonymized in the same way. But while the diameter does this only by looking at pairs of records in M, the generalization costs do this by looking simultaneously at all records in M and computing the information loss that their closure entails. This simple difference turns out to be very important, as we show below.

Before moving on, we state the following basic lemma that plays a significant role in our analysis.

Lemma 2. *Assume that all collections of subsets, $\overline{A}_j$, $1 \leq j \leq r$, are proper. Then the generalization costs $d(\cdot)$, (3) and (4), are sub-additive in the sense that for all $S, T \subseteq A_1 \times \cdots \times A_r$,*

$$S \cap T \neq \emptyset \text{ implies that } d(S \cup T) \leq d(S) + d(T) \,. \tag{7}$$

5.2 Covers, Clusterings, k-Anonymizations and Their Generalization Cost

As noted earlier, any k-anonymization of D defines a clustering of D. Without loss of generality, we may assume that all clusters are of sizes between k and $2k-1$; indeed, owing to monotonicity, any cluster of size greater than $2k$ may be split into clusters of sizes in the range $[k, 2k-1]$ without increasing the amount of information loss due to k-anonymization. Let:

1. $\mathcal{G}$ be the family of all k-anonymizations of D, where the corresponding clusters are of sizes in the range $[k, 2k-1]$.
2. Γ be the family of all covers of D by subsets of sizes in the range $[k, 2k-1]$.
3. $\Gamma^0 \subset \Gamma$ be the family of all covers in Γ that are clusterings (or partitions); namely, all covers in Γ consisting of non-intersecting subsets.

There is a natural one-to-one correspondence between $\mathcal{G}$ and Γ^0.

Hereinafter, Π denotes either the entropy measure of loss of information, $\Pi = \Pi_{\mathrm{e}}$, or the monotone entropy measure of loss of information, $\Pi = \Pi_{\mathrm{me}}$. The corresponding generalization cost is then denoted by $d(\cdot)$ (namely, $d(\cdot) = d_{\mathrm{e}}$ if $\Pi = \Pi_{\mathrm{e}}$ and $d(\cdot) = d_{\mathrm{me}}$ if $\Pi = \Pi_{\mathrm{me}}$).

Given a cover $\gamma \in \Gamma$, we define its generalization cost as $d(\gamma) = \sum_{S \in \gamma} d(S)$.

Theorem 2. *Let $\hat{\gamma}$ be a cover that achieves minimal generalization cost $d(\cdot)$ in Γ. Let $g \in \mathcal{G}$ be a k-anonymization and let $\gamma^0 \in \Gamma^0$ be its corresponding clustering. Then*

$$\Pi(D, g(D)) \leq \frac{2d(\gamma^0)}{d(\hat{\gamma})} \cdot OPT(D), \tag{8}$$

where

$$OPT(D) := \min_{g \in \mathcal{G}} \Pi(D, g(D)). \tag{9}$$

5.3 Approximating Optimal k-Anonymization

Our approximation algorithm follows the algorithm of [19]. It has two phases, as described hereinafter.

Phase 1: Producing a cover. Let $\hat{\gamma}$ be a cover that minimizes $d(\cdot)$ in Γ. In the first phase of the algorithm we execute the greedy algorithm for approximating the WEIGHTED SET COVER problem [15].

1. Set $\mathcal{C}$ to be the collection of all subsets of D with cardinality in the range $[k, 2k-1]$.
2. Set $\gamma = \emptyset$ and $E = \emptyset$.
3. While $E \neq D$ do:
   - For each $S \in \mathcal{C}$ compute the ratio $r(S) = d(S)/|S \cap (D \setminus E)|$.
   - Choose S that minimizes $r(S)$.
   - $E = E \cup S$, $\gamma = \gamma \cup \{S\}$, $\mathcal{C} = \mathcal{C} \setminus \{S\}$.
4. Output γ.

Since the greedy algorithm for the WEIGHTED SET COVER problem has logarithmic approximation guarantee (see, e.g., [8]), the result of that phase is a cover $\gamma \in \Gamma$ for which $d(\gamma) \leq (1 + \ln 2k) d(\hat{\gamma})$.

Phase 2: Translating the cover into a k-anonymization. In the second phase we translate the cover $\gamma \in \Gamma$ to a clustering $\gamma^0 \in \Gamma^0$ and then to its corresponding k-anonymization $g \in \mathcal{G}$. The translation procedure works as follows:

1. Input: $\gamma = \{S_1, \ldots, S_t\}$, a cover of $D = \{R_1, \ldots, R_n\}$.
2. Set $\gamma^0 = \gamma$.

3. Repeat until the cover γ^0 has no intersecting subsets:
   - Let $S_j, S_\ell \in \gamma^0$ be such that $S_j \cap S_\ell \neq \emptyset$ and let R be a record in D that belongs to $S_j \cap S_\ell$.
   - If $|S_j| > k$ set $S_j = S_j \setminus \{R\}$.
   - Else, if $|S_\ell| > k$ set $S_\ell = S_\ell \setminus \{R\}$.
   - Else (namely, if $|S_j| = |S_\ell| = k$) remove S_ℓ from γ^0 and set $S_j = S_j \cup S_\ell$.
4. Output the following k-anonymization: For $i = 1, \ldots, n$, look for $S_j \in \gamma^0$ such that $R_i \in S_j$ and then set $g(D)_i = \overline{S}_j$.

Theorem 3. *The k-anonymization g that is produced by the above described algorithm satisfies*

$$\Pi(D, g(D)) \leq 2(1 + \ln 2k) \cdot OPT(D)\,, \tag{10}$$

where $OPT(D)$ is the cost of an optimal k-anonymization, (9).

The corresponding result in [19] is Theorem 4.1 there, according to which the approximation algorithm achieves an approximation factor of $3k \cdot (1 + \ln 2k)$. Aggarwal et al. proposed an improved approximation algorithm that achieves an $O(k)$ approximation factor [1, Theorem 5]. The approximation algorithms in both [19] and [1] were based on the so-called *graph representation*. In that approach, the records of D are viewed as nodes of a complete graph, where the weight of each edge (R_i, R_j) is the generalization cost of the set $\{R_i, R_j\}$. Both algorithms work with such a graph representation and find the approximate k-anonymization based only on the information that is encoded in that graph. Such an approach is limited since it uses only the distances between pairs of nodes. In [1] it was shown that using the graph representation it is impossible to achieve an approximation ratio that is better than $\Theta(k)$.

We were able to offer the significantly better $O(\ln k)$ approximation ratio by breaking out of the graph representation framework. As explained in Section 5.1, our cost function $d(\cdot)$ is defined for sets of records, rather than pairs of records. Hence, it represents *volume* rather than a *diameter*. This upgrade from the graph representation to a hypergraph representation enabled the improvement from a linear approximation ratio to a logarithmic one.

It should be noted that our improved approximation algorithm works also with the tree measure, if we modify the definition of the generalization cost, Definition 10, to be consistent with that measure. Such a modified generalization cost is clearly monotone, (5), and sub-additive, (7), whence all of our claims hold also for that cost. The algorithm described in this section runs in time $O(n^{2k})$. The exponential dependence of the running time on k is due to the fact that we examine all subsets of records of D with cardinalities between k and $2k - 1$.

5.4 A Fully Polynomial Approximating Algorithm

Here we describe briefly (due to space limitations) the algorithm of Aggarwal et al. [1], and concentrate on the necessary modifications that are required in order to make it work for our entropy measure.

The algorithm starts by considering the graph representation $G = (V, E)$ of the database D. This is a complete weighted graph, where $V = D = \{R_1, \ldots, R_n\}$, and the edge $e_{i,j} = \{R_i, R_j\} \in E$ has weight $w(e_{i,j}) = d(\{R_i, R_j\})$, where $d(\cdot)$ is the generalization cost by the entropy measure, (3). Let $\mathcal{F} = \{\mathcal{T}_1, \ldots, \mathcal{T}_s\}$ be a spanning forest of G. If all trees in that forest are of size at least k then that forest induces a k-anonymization of D, denoted $g_{\mathcal{F}}$ (namely, all records in the tree $\mathcal{T}_\ell$ are replaced by the closure of that tree, $\overline{\mathcal{T}_\ell}$). The charge of each node with respect to $g_{\mathcal{F}}$ is defined as $c(R_i, g_{\mathcal{F}}) = d(\mathcal{T}_{j(i)})$, where $d(\cdot)$ is the generalization cost by the measure Π (that could be either the entropy measure, Π_{e}, or the monotone entropy measure, Π_{me}). The generalization cost of $g_{\mathcal{F}}$ is then

$$\Pi(D, g_{\mathcal{F}}(D)) = \sum_{i=1}^{n} c(R_i, g_{\mathcal{F}})\,. \tag{11}$$

Theorem 4. *Let $OPT = OPT(D)$ be the cost of an optimal k-anonymization of D with respect to the measure of loss of information, Π, and let L be an integer such that $L \geq k$. Let $\mathcal{F} = \{\mathcal{T}_1, \ldots, \mathcal{T}_s\}$ be a spanning forest of G whose total weight is at most OPT and in which each of the trees is of size in the range $[k, L]$. Then the corresponding k-anonymization, $g_{\mathcal{F}}$, is an L-approximation for the optimal k-anonymization, i.e.,*

$$\Pi(D, g_{\mathcal{F}}(D)) \leq L \cdot OPT\,.$$

The algorithm then proceeds in two stages:

Stage 1: Create a spanning forest $\mathcal{F} = \{\mathcal{T}_1, \ldots, \mathcal{T}_s\}$ whose total weight is at most OPT (the cost of an optimal k-anonymization) and in which all trees are of size at least k.

Stage 2: Compute a decomposition of this forest such that each component has size in the range $[k, L]$ for $L = \max\{2k - 1, 3k - 5\}$.

Both stages are described in detail in [1]. In view of Theorem 4, this algorithm achieves an approximation ratio of $O(k)$. Its analysis, to a large extent, is independent of the underlying measure of loss of information that determines the weight of the edges. Furthermore, it is a fully polynomial algorithm whose running time is $O(kn^2)$.

Acknowledgements. The authors thank Jacob Goldberger who proposed the *monotone entropy measure* as a modification of the entropy measure that respects monotonicity.

References

1. Aggarwal, G., Feder, T., Kenthapadi, K., Motwani, R., Panigrahy, R., Thomas, D., Zhu, A.: Anonymizing tables. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363. Springer, Heidelberg (2004)
2. Aggarwal, G., Mishra, N., Pinkas, B.: Secure computation of the kth-ranked element. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027. Springer, Heidelberg (2004)

3. Agrawal, D., Aggarwal, C.: On the design and quantification of privacy preserving data mining lgorithms. In: PODS (2001)
4. Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: SIGMOD (2000)
5. Agrawal, R., Srikant, R., Thomas, D.: Privacy preserving OLAP. In: SIGMOD (2005)
6. Blum, A., Dwork, C., McSherry, F., Nissim, K.: Practical privacy: The SuLQ framework. In: PODS (2005)
7. Chawla, S., Dwork, C., McSherry, F., Smith, A., Wee, H.: Toward privacy in public databases. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378. Springer, Heidelberg (2005)
8. Chvatal, V.: A greedy heuristic for the set-covering problem. Mathematics of Operations Research 4(3), 233–235 (1979)
9. DeWaal, A.G., Willenborg, L.C.R.J.: Information loss through global recoding and local suppression. Netherlands Official Statistics, Special issue on SDC 14, 17–20 (1999)
10. Dinur, I., Nissim, K.: Revealing information while preserving privacy. In: PODS (2003)
11. Dwork, C., Nissim, K.: Privacy-preserving data mining on vertically partitioned databases. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152. Springer, Heidelberg (2004)
12. Evfimievski, A., Gehrke, J., Srikant, R.: Limiting privacy breaches in privacy preserving data mining. In: PODS (2003)
13. Freedman, M., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027. Springer, Heidelberg (2004)
14. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: STOC (1987)
15. Johnson, D.S.: Approximation algorithms for combinatorial problems. JCSS 9, 256–278 (1974)
16. Kenthapadi, K., Mishra, N., Nissim, K.: Simulatable auditing. In: PODS (2005)
17. Kleinberg, J., Papadimitriou, C., Raghavan, P.: Auditing boolean attributes. JCSS 6, 244–253 (2003)
18. Lindell, Y., Pinkas, B.: Privacy preserving data mining. Journal of Cryptology 15(3), 177–206 (2002)
19. Meyerson, A., Williams, R.: On the complexity of optimal k-anonymity. In: PODS (2004)
20. Samarati, P.: Protecting respondent's privacy in microdata release. TKDE 13, 1010–1027 (2001)
21. Samarati, P., Sweeney, L.: Generalizing data to provide anonymity when disclosing information (abstract). In: PODS (1998)
22. Sweeney, L.: k-Anonymity: A model for protecting privacy. International Journal on Uncertainty, Fuzziness and Knowledge-based Systems 10(5), 557–570 (2002)
23. Willenborg, L., DeWaal, T.: Elements of Statistical Disclosure Control. Springer, Heidelberg (2001)
24. Yao, A.: How to generate and exchange secrets. In: FOCS (1986)

A Quasi-PTAS for Profit-Maximizing Pricing on Line Graphs*

Khaled Elbassioni[1], René Sitters[2], and Yan Zhang[3]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
elbassio@mpi-inf.mpg.de
[2] Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
r.sitters@tue.nl
[3] Department of Computer Science and Engineering,
Hong Kong University of Science and Technology, Hong Kong, China
cszy@cse.ust.hk

Abstract. We consider the problem of pricing items so as to maximize the profit made from selling these items. An instance is given by a set E of n items and a set of m clients, where each client is specified by one subset of E (the bundle of items he/she wants to buy), and a budget (valuation), which is the maximum price he is willing to pay for that subset. We restrict our attention to the model where the subsets can be arranged such that they form intervals of a line graph. Assuming an unlimited supply of any item, this problem is known as *the highway problem* and so far only an $O(\log n)$-approximation algorithm is known. We show that a PTAS is likely to exist by presenting a quasi-polynomial time approximation scheme. We also combine our ideas with a recently developed quasi-PTAS for the unsplittable flow problem on line graphs to extend this approximation scheme to the limited supply version of the pricing problem.

1 Introduction

Suppose you have a set of items to sell and you have complete information of your possible clients, i.e., you know what each client wants to buy and how much he (she) is willing to pay for this. How do you set your prices such that your total profit is maximized? If you price the items cheap, then many people will buy from you but, of course, at a low price. If you price too high, then only few people are willing to pay the price. This kind of pricing problems appear more and more in pricing mechanisms over the Internet. The seller collects statistical data and adjusts the prices online depending on the set of potential buyers. Recently, a number of papers appeared on the computational complexity of these problems [1,2,4,5,6,7,8,9].

Pricing problems of this type are generally very hard to solve because of their non-linear nature: each client will either buy the whole bundle of items he is

* The work of the third author was partially supported by Hong Kong RGC CERG grant HKUST6312/04E.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 451–462, 2007.

interested in, or not buy at all; there is no solution in between (i.e. the client is not interested in buying a strict subset of the bundle). We restrict our attention to the so called *single minded* clients, i.e., each client is only interested in a single subset of the items. If there is an *unlimited* supply of any item, then the client will buy his preferred subset if and only if the total price of the items in the subset is no more than his budget (such a pricing is said to be *envy-free* since, given the pricing, no client would prefer to be assigned a different bundle). Guruswami et al. [8] show that this problem is already APX-hard if all budgets are equal to one and each client wants to buy a single pair of items. They also give an $O(\log n + \log m)$-approximation algorithm, where n and m are, respectively, the numbers of items and clients. Efficient (approximation) algorithms are known for several variants in which constraints are placed on the numbers in the input. For example, Hartline and Koltun [9] give a PTAS for the case when the number of items is constant, and Balcan and Blum [2] give an $O(k)$-approximation algorithm under the restriction that the bundle of any client contains at most k items.

In this paper we do not impose any restriction on the sizes, prices, capacities or budgets but only restrict the system of subsets (bundles). More precisely, we assume that the subsets can be arranged such that they form intervals of a line graph. This special case is known as the *highway problem*, and recently Balcan and Blum [2] gave an $O(\log n)$-approximation algorithm improving over the $O(\log n + \log m)$ algorithm for the general problem, and an $O(1)$-approximation when all intervals are of almost the same length. NP-hardness of the highway problem was shown by Briest and Krysta [4], and Guruswami et al. [8] gave a polynomial-time exact algorithm for the case when all budgets are bounded by a constant, and a pseudo-polynomial-time algorithm for the case when all intervals are of constant length and all budgets are integral.

In Section 5 we consider the *capacitated* variant in which there is only a limited supply of any item. Hence, we have to set prices as well as to find a subset of the clients such that the supply constraints are satisfied. This problem has some similarities with the *unsplittable flow problem* on line graphs for which a quasi-PTAS was developed only recently by Bansal et al [3]. To solve the limited supply case, we consider the following generalization which may be of independent interest: Assume that there are capacities on the edges. Assume further that each buyer I wants to route a demand of $\rho(I)$ along the interval I, and has budget $B(I)$. If a buyer can route his demand, he will purchase the path, provided its price is within his budget. Then the question is how to price the edges such that the total routed demand on each edge, from buyers that can afford to route their demands, does not exceed the capacity of the edge and the total profit is maximized[1]. The limited supply version of the highway problem corresponds to the case with unit demands. Just as the quasi-PTAS for the unsplittable flow problem [3], the technique used here is standard for

[1] Note, however, that the resulting pricing may not be envy-free in this case, since some of the clients who can afford to purchase their paths, may not be able to do so because there might not be enough capacity on the edges.

deriving polynomial time approximation schemes: first rounding, then restricting the search space further, and finally applying dynamic programming. However, for both problems the second step is non-trivial. In the dynamic programming we do not enumerate directly over all possible (rounded) price vectors but, instead, over systems of linear inequalities on the price vectors. For each consistent set of inequalities we can easily derive the value of the solution with a feasible price vector.

2 The Setting

Let $V = \{0, 1, \ldots, n\}$ and $E = \{e_1, \ldots, e_n\}$, with $e_i = \{i-1, i\}$, for $i = 1, \ldots, n$. We assume that we are given a (multi)set of intervals (bundles) $\mathcal{I} = \{I_1, \ldots, I_m\}$, defined on the set of edges (items) E, where $I_j = [s_j, t_j] \stackrel{\text{def}}{=} \{\{s_j, s_j+1\}, \{s_j+1, s_j+2\} \ldots, \{t_j-1, t_j\}\} \subseteq E$. For $I \in \mathcal{I}$, we denote by $B(I) \in \mathbb{R}_+$ the *budget* of interval I. In the *highway problem*, denoted henceforth by Hp, the objective is to assign a price $p(e) \in \mathbb{R}_+$ for each edge $e \in E$, and to find a subset $\mathcal{J} \subseteq \mathcal{I}$, so as to maximize

$$\sum_{I \in \mathcal{J}} p(I) \tag{1}$$

subject to the budget constraints

$$p(I) \leq B(I), \quad \text{for all } I \in \mathcal{J}, \tag{2}$$

where, $p(I) = \sum_{e \in I} p(e)$. When the number of copies of each edge $e \in E$, available for purchase, is limited by a given number, the problem will be called the *capacitated highway problem*. As mentioned in the introduction, this version of the problem can be cast as an instance of the following more general problem, which we call the *unsplittable flow pricing problem* and denote by Ufp: Let $c(e) \in \mathbb{R}_+$, be the *available supply* or *capacity of edge* e, and $\rho(I)$ be the demand of interval $I \in \mathcal{I}$. The requirement is to assign a price $p(e) \in \mathbb{R}_+$ for each edge $e \in E$, and to find a subset $\mathcal{J} \subseteq \mathcal{I}$, so as to maximize (1) subject to (2) and the capacity constraints

$$\sum_{I \in \mathcal{J}: e \in I} \rho(I) \leq c(e), \quad \text{for all } e \in E. \tag{3}$$

In the following sections, we denote by $p^* : E \mapsto \mathbb{R}_+$ the optimal set of prices and by Opt $\subseteq \mathcal{I}$ the set of intervals purchased in the optimum solution. It is easy to see that specifying any one of these two sets completely determines the other, and thus any of the two is enough to completely describe the optimal solution. For a subset of intervals $\mathcal{I}' \subseteq \mathcal{I}$, and a price function $p : E \mapsto \mathbb{R}_+$, we denote by $p(\mathcal{I}') = \sum_{I \in \mathcal{I}'} p(I)$ the total price of intervals in $\mathcal{I}'$. Similar notation will be also used for the function $\rho(\cdot)$.

For an edge $e = \{u-1, u\} \in E$ and a (multi)set of intervals $\mathcal{I}$ on E, denote respectively by $\mathcal{I}_L(e)$, $\mathcal{I}_R(e)$, and $\mathcal{I}[e]$ the subsets of intervals of $\mathcal{I}$ that lie to the left of e, lie to the right of e, and span e, that is

$$\mathcal{I}_L(e) = \{[s,t] \in \mathcal{I} : \ t \le u-1\},$$
$$\mathcal{I}_R(e) = \{[s,t] \in \mathcal{I} : \ s \ge u\},$$
$$\mathcal{I}[e] = \{[s,t] \in \mathcal{I} : \ s \le u-1 < u \le t\}.$$

Denote by $E_L(e)$ and $E_R(e)$ the sets of edges that lie to the left and right of edge $e \in E$, respectively:

$$E_L(e) = \{\{x-1,x\} \in E : \ x \le u-1\} \text{ and } E_R(e) = \{\{x-1,x\} \in \mathcal{I} : \ x-1 \ge u\}.$$

3 Rounding the Instance

Let $\epsilon > 0$ be a given constant. We may assume without loss of generality that for every vertex $u \in V$, there is an interval $I \in \mathcal{I}$ beginning at that point (otherwise we can merge indistinguishable edges). In particular, $n \le 2m$ can be assumed. Denote by $B_{max} = \max\{B(I) : I \in \mathcal{I}\}$ the maximum budget in the instance. Clearly, $B_{max} \le p^*(\textsc{Opt})$ since we can assign a price of $B_{max}/|I_{max}|$ for every edge of the interval of maximum budget I_{max}, where $|I_{max}|$ is the length of I_{max}.

Consider an optimal price function p^*. We obtain a new price function p' by rounding *down* $p^*(e)$, for each $e \in E$, to the closest multiple of $\epsilon B_{max}/(nm)$. Then the total reduction in profit is $p^*(\textsc{Opt}) - p'(\textsc{Opt}) \le \epsilon B_{max} \le \epsilon p^*(\textsc{Opt})$, i.e. $p'(\textsc{Opt}) \ge (1-\epsilon)p^*(\textsc{Opt})$. Now we scale all budgets and prices in p' by $nm(1+\epsilon)/(B_{max}\epsilon)$ (this does not change the optimal solution), and assume, at the loss of factor of $(1-\epsilon)$ of the optimal, that all prices belong to the set $\{0, 1+\epsilon, 2(1+\epsilon), \ldots, P(1+\epsilon)\}$, where $P \stackrel{\text{def}}{=} \frac{mn}{\epsilon}$. Note that we still have $p'(I) \le B(I)$ for all $I \in \textsc{Opt}$.

By dividing the prices further by $(1+\epsilon)$, we obtain a set of prices $\tilde{p} : E \mapsto \mathbb{R}_+$, for which can also assume that every interval $I \in \textsc{Opt}$ has $\tilde{p}(I) = p'(I)/(1+\epsilon) \le B(I)/(1+\epsilon)$. The total profit of such solution is $\tilde{p}(\textsc{Opt}) \ge \frac{1-\epsilon}{1+\epsilon}\textsc{Opt} \ge (1-2\epsilon)\textsc{Opt}$. We summarize the above facts in the following proposition.

Proposition 1. *Let p^* be an optimal solution for a given instance of* HP*, and $\epsilon > 0$ be a given constant. Then there exists a pricing $\tilde{p} : E \mapsto \mathbb{R}_+$ for which*

(i) $\tilde{p}(e) \in \{0, 1, \ldots, P\}$, for every $e \in E$, where $P = nm/\epsilon$,
(ii) $\tilde{p}(I) \le \frac{B(I)}{1+\epsilon}$, for every $I \in \textsc{Opt}$, and
(iii) $\tilde{p}(\textsc{Opt}) \ge (1-2\epsilon)p^*(\textsc{Opt})$.

We shall call any pricing $\tilde{p}$, satisfying the conditions of Proposition 1, an ϵ-*optimal pricing*.

4 The Highway Problem

In this case, we assume that $c(e) = \infty$ for all $e \in E$. Given a price function $p : E \mapsto \mathbb{R}_+$ and an edge $e^* = \{u^*-1, u^*\} \in E$, the *accumulative price* at any edge $e = \{x-1, x\} \in E$ with respect to e^* is defined as $p([x-1, u^*])$ if

e is to the left of e^* (i.e. if $x \leq u^* - 1$) and $p([u^*, x])$ if e is to the right of e^* (i.e. if $x - 1 \geq u^*$). Obviously, starting from e^*, these accumulative prices form monotonically increasing functions to the left and right of e^*.

In this section we prove the following statement.

Theorem 1. *There is a quasi-polynomial time approximation scheme for the highway problem.*

Our QPTAS is based essentially on the same divide and conquer strategy used in [3]. It starts by picking an edge in the middle and guesses the points at which the ϵ-optimal accumulative prices increase by factors of $(1+\epsilon)$ relative to that middle edge. Having guessed such "increment points", the algorithm picks a *superset* of the optimal set of intervals containing the middle edge, then recurses independently on the two subproblems to the left and right of the middle edge. In the following, we fix $k = \lceil \log(nP)/\log(1+\epsilon) \rceil + 1$.

Definition 1. (ϵ-Relative pricings) *Let $e^* = [u^* - 1, u^*] \in E$ be a given edge of E, and $1 \leq k_1, k_2 \leq k$ be given integers. A selection of $k_1 + k_2 + 2$ points $u_1, \ldots, u_{k_1}, u_{k_1+1}, u'_1, \ldots, u'_{k_2}$, $u'_{k_2+1} \in V$, and $k_1 + k_2 + 2$ values $p_1, \ldots, p_{k_1}$, p_{k_1+1}, $p'_1, \ldots, p'_{k_2}, p'_{k_2+1} \in \{0, 1, \ldots, P\}$, such that*

1. $u_{k_1+1} \leq u_{k_1} < u_{k_1-1} < \cdots < u_1 \leq u^* < u'_1 < u'_2 < \cdots < u'_{k_2} \leq u_{k_2+1}$,
2. $p_j \geq (1+\epsilon)p_{j-1}$, *for* $j = 2, \ldots, k_1$, *and* $p'_j \geq (1+\epsilon)p'_{j-1}$, *for* $j = 2, \ldots, k_2$,
3. $p_{k_1+1} \geq p_{k_1}$ *and* $p'_{k_2+1} \geq p'_{k_2}$.

is said to be an ϵ-relative pricing *of E w.r.t. e^*, denoted by* $(u^*, u_1, \ldots, u_{k_1+1}, u'_1, \ldots, u'_{k_2+1}, p_1, \ldots, p_{k_1+1}, p'_2, \ldots, p'_{k_2+1})$.

See Figure 1 for an example. The total number of possible ϵ-relative pricings with respect to a given edge $e^* \in E$ is at most

$$L = [n(1+P)]^{2k+2} = \left(n + \frac{n^2 m}{\epsilon}\right)^{2\lceil \log \frac{n^2 m/\epsilon}{\log(1+\epsilon)} \rceil + 4}, \tag{4}$$

which is $m^{O(\log(m))}$ for every fixed $\epsilon > 0$.

Let $R = (u^*, u_1, \ldots, u_{k_1+1}, u'_1, \ldots, u'_{k_2+1}, p_1, \ldots, p_{k_1+1}, p'_2, \ldots, p'_{k_2+1})$ be an ϵ-relative pricing w.r.t. an edge e^*. Given an interval $I = [s, t] \in \mathcal{I}$, with $e^* \in I$, we associate a value $v(I, R)$ to I, defined with respect to R as follows. Let $j(I), l(I)$ be respectively the smallest and largest indices such that $e_{i_{j(I)}}, e'_{i_{l(I)}} \in I$, i.e. $j(I) = \min\{i : u_i - 1 \geq s\}$ and $l(I) = \max\{i : u'_i \leq t\}$ (see Figure 1). Then, for $I \in \mathcal{I}$, define

$$v(I, R) = p_{j(I)} + p'_{l(I)}.$$

(If either $j(I)$ or $l(I)$ does not exist, the corresponding value $p_{j(I)}$ or $p'_{l(I)}$ is set to 0.) For a subset of intervals $\mathcal{I}' \subseteq \mathcal{I}$, we define as usual, $v(\mathcal{I}', R) = \sum_{I \in \mathcal{I}'} v(I, R)$.

Definition 2. (Consistent pricings) *Let $R = (u^*, u_1, \ldots, u_{k_1+1}, u'_1, \ldots, u'_{k_2+1}$, $p_1, \ldots, p_{k_1+1}, p'_2, \ldots, p'_{k_2+1})$ be an ϵ-relative pricing of E w.r.t. an edge $e^* \in E$, and $p : E \mapsto \mathbb{R}_+$ be a price function. We say that p and R are* consistent *if*

(C1) $p([u_j - 1, u^*]) = p_j$ *for* $j = 1, \ldots, k_1 + 1$, *and* $p([u^*, u'_j]) = p'_j$ *for* $j = 1, \ldots, k_2 + 1$,
(C2) $p([u_j, u^*]) \leq (1+\epsilon)p_{j-1}$ *for* $j = 2, \ldots, k_1+1$, *and* $p([u^*, u'_{j-1}]) \leq (1+\epsilon)p'_{j-1}$ *for* $j = 2, \ldots, k_2 + 1$.

It follows that for any ϵ-relative pricing R w.r.t. an edge $e^* \in E$ and any $q : E \mapsto \mathbb{R}_+$ with which R is consistent, we have

$$v(I, R) \leq q(I) \leq (1 + \epsilon)v(I, R) \text{ for all } I \in \mathcal{I}[e^*]. \tag{5}$$

This property will be used crucially in our analysis.

Lemma 1. *Let* $p : E \mapsto \mathbb{R}_+$ *be a pricing for a given instance of* HP *and* $e^* = [u^* - 1, u^*]$ *be an arbitrary edge. Then there exists an* ϵ*-relative pricing of* E *w.r.t.* e^*, *that is consistent with* p.

Proof. Assume that $E = \{\{0, 1\}, \{1, 2\}, \ldots, \{n-1, n\}\}$. We define a selection as follows (see Figure 1). Let $u_1 = \max\{u \leq u^* : \ p([u - 1, u]) > 0\}$, $u'_1 = \min\{u > u^* : \ p([u - 1, u]) > 0\}$, and set $p_1 = p([u_1 - 1, u_1])$ and $p'_1 = p([u'_1 - 1, u'_1])$. For $j = 2, 3, \ldots$, let $u_j = \max\{u \leq u_{j-1} : \ p([u - 1, u^*]) > (1 + \epsilon)p_{j-1}\}$, and set $p_j = p([u_j - 1, u^*])$. The highest index j for which this iteration can be done will the value of k_1. Similarly, we define k_2, and for $j = 2, \ldots, k_2$, let $u'_j = \min\{u > u'_{j-1} : \ p([u^*, u]) > (1 + \epsilon)p'_{j-1}\}$ and set $p'_j = p([u^*, u'_j])$. Since $nP \geq p_{k_1} \geq (1+\epsilon)^{k_1-1}p_1 \geq (1+\epsilon)^{k_1-1}$, we get $k_1 \leq k$. Similarly, $k_2 \leq k$. Finally, we let $u_{k_1+1} = 1$, $u_{k_2+1} = n$, $p_{k_1+1} = p([0, u^*])$ and $p'_{k_2+1} = p([u^*, n])$. □

With every ϵ-relative pricing R, we can associate a system of linear inequalities, denoted by $S(R)$, on a set of E variables $\{p(e) : \ e \in E\}$, consisting of the constraints $(C1)$ and $(C2)$ together with the non-negativity constraints $p(e) \geq 0$. For two systems of inequalities S_1, S_2, we denote by $S_1 \wedge S_2$ the system obtained by combining their inequalities.

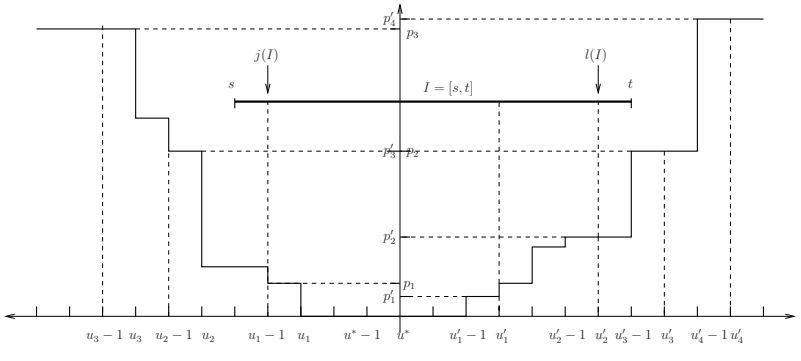


Fig. 1. ϵ-relative pricings w.r.t. to $[u^* - 1, u^*]$: To the left is the plot of $p([u - 1, u^*])$ for $u < u^*$, and to the right is the one for $p([u^*, u])$ for for $u \geq u^*$. The indices $j(I)$ and $l(I)$ for a given interval $I = [s, t]$ are also shown.

The algorithm is shown in Figure 2. It is initially called with an empty $\mathcal{S}$. The procedure iterates over all ϵ-relative pricings R, consistent with $\mathcal{S}$, w.r.t. to the middle edge e^*, then recurses on the subsets of intervals to the left and right of e^*. In line 4, we always insure that $S(R) \wedge \mathcal{S}$ is defined on variables $\{p(e) : e \in E_L(e^*)\}$. Similarly, in line 5, we always insure that $S(R) \wedge \mathcal{S}$ is defined on variables $\{p(e) : e \in E_R(e^*)\}$. This is necessary for the induction proof in Lemma 3 below to work, and can be maintained as follows. Assume at some iteration that $e^* = \{u^* - 1, u^*\}$ and $E = \{\{i, i+1\}, \{i+1, i+2\}, \ldots, \{r-1, r\}\}$. By the way we defined ϵ-relative pricings, we note that the following invariant holds throughout the algorithm:

(I) Every constraint in $\mathcal{S}$ is defined on an interval starting at $\{i, i+1\}$, or ending at $\{r-1, r\}$.

This is trivially true initially, and can be shown by induction for any iteration: (i) all the constrains of the form $(C1)$ or $(C2)$ defined on an edge of $E_L(e^*)$ are of the form $p([u, u^*]) \leq w$ (or $p([u, u^*]) = w$), where $u \in e \in E_L(e^*)$ and $w \in \mathbb{R}_+$. Note that $S(R)$ implies a constraint of the form $p(e^*) = q$ for some $q \in \mathbb{R}_+$, and thus any constraint $p([u, u^*]) \leq w$ can be reduced to the equivalent one $p([u, u^* - 1]) \leq w - q$. (ii) Any constraint in $\mathcal{S}$ of the form $p([i, u]) \leq w$, where $u > u^*$ and $w \in \mathbb{R}_+$, can be safely removed since $S(R) \wedge \mathcal{S}$ is feasible and $S(R)$ already contains the constraint $p([i, u^*]) = w'$, for some $w' \leq w$. (iii) $S(R)$ contains also the constraint $p([u^*, r]) = w$, for some $w \in \mathbb{R}_+$, and thus any constraint of $\mathcal{S}$ the form $p([u, r]) \leq w'$, where $u < u^*$ and $w' \geq w + q$ can be replaced by $p([u, u^* - 1] \leq w' - w - q$. Thus we conclude that the (new equivalent) system $\mathcal{S} \wedge S(R)$ satisfies the invariant on $E_L(e^*)$. A similar reasoning also shows the invariant is satisfied on $E_R(e^*)$.

When the procedure returns, we get two price functions $p_1 : E_L(e^*) \mapsto \mathbb{R}_+$ and $p_2 : E_R(e^*) \mapsto \mathbb{R}_+$. Let q be the value assigned to e^* by $S(R)$. We define a price function $p : E \mapsto \mathbb{R}_+$ on E as follows

$$p(e) = \begin{cases} p_1(e), & \text{if } e \in E_L(e^*) \\ p_2(e), & \text{if } e \in E_R(e^*) \\ q, & \text{If } e = e^*, \end{cases} \tag{6}$$

the procedure also returns a set of intervals which can be purchased under the returned price function p.

Lemma 2. *Algorithm* HP *runs in quasi-polynomial time in m, for any fixed $\epsilon > 0$.*

Proof. The number of possible ϵ-relative pricing is at most L, given in (4). This gives the recurrence

$$T(m) \leq \text{poly}(m) + 2L \cdot T(\frac{m}{2}).$$

for the running time. Thus $T(m) \leq L^{\log m + 1}\,\text{poly}(m)$ and the lemma follows. □

Lemma 3. *Algorithm* HP *returns a price function p and a set of intervals $\mathcal{J}$ such that $p(\mathcal{J}) \geq (1 - 3\epsilon)p^*(\text{OPT})$, for any $\epsilon > 0$.*

Algorithm HP$(\mathcal{I}, E, \mathcal{S})$**:**
Input: A subset of intervals $\mathcal{I}$ defined on E, and a feasible system of inequalities $\mathcal{S}$
Output: A price function $p : E \mapsto \mathbb{R}_+$ and a subset $\mathcal{J} \subseteq \mathcal{I}$ s.t. $p(I) \leq B(I)\ \forall I \in \mathcal{J}$

1. **if** $|\mathcal{I}| = 0$, **then return** $(p, \emptyset)$, where p is any feasible solution of $\mathcal{S}$
2. let e^* be an edge of E such that $|\mathcal{I}_L(e^*)| \leq m/2$ and $|\mathcal{I}_R(e^*)| \leq m/2$
3. **for** every ϵ-relative pricing R w.r.t. e^* for which $\mathcal{S} \wedge S(R)$ is feasible **do**
4. $\quad (p_1, \mathcal{J}_1) \leftarrow \text{HP}(\mathcal{I}_L(e^*), E'_L(e^*), \mathcal{S} \wedge S(R))$
5. $\quad (p_2, \mathcal{J}_2) \leftarrow \text{HP}(\mathcal{I}_R(e^*), E'_R(e^*), \mathcal{S} \wedge S(R))$
6. $\quad$ let p be the price function defined by (6)
7. $\quad \mathcal{K} \leftarrow \{I \in \mathcal{I}[e^*] :\ v(I, R) \leq B(I)/(1+\epsilon)\}$
8. $\quad \mathcal{J} \leftarrow \mathcal{K} \cup \mathcal{J}_1 \cup \mathcal{J}_2$
9. $\quad$ record $(p, \mathcal{J})$
10. **return** the recorded solution with largest $p(\mathcal{J})$ value

Fig. 2. The dynamic program for computing ϵ-approximate prices

Proof. Fix an optimal solution Opt and an optimal price p^*, and let $\tilde{p}$ be an ϵ-optimal price. We prove by induction the following statement:

(H) Let $(\mathcal{I}, E, \mathcal{S})$ be the input to the algorithm and let $\mathcal{J}' \subseteq \mathcal{I}$ and $p' : E \mapsto \mathbb{R}_+$ be such that $p'(I) \leq B(I)/(1+\epsilon)$ for all $I \in \mathcal{J}'$, and p' satisfies $\mathcal{S}$. Then the algorithm returns a pricing $p : E \mapsto \mathbb{R}_+$ and a subset $\mathcal{J} \subseteq \mathcal{I}$ such that (i) p satisfies $\mathcal{S}$, (ii) $p(I) \leq B(I)$ for all $I \in \mathcal{J}$, and (iii) $p(\mathcal{J}) \geq p'(\mathcal{J}')/(1+\epsilon)$.

This can be used to prove the statement of theorem by setting $\mathcal{J}' \leftarrow$ Opt and $p' \leftarrow \tilde{p}$. Then it will follow by (H) and Proposition 1 that the algorithm returns a subset of intervals $\mathcal{J}$ and a pricing p such that all intervals in $\mathcal{J}$ are purchased, and their total price is at least $\tilde{p}(\text{Opt})/(1+\epsilon) \geq (1-3\epsilon)p^*(\text{Opt})$.

Now we prove (H). Let e^* be the middle edge. By Lemma 1, there is an ϵ-relative pricing R w.r.t e^*, consistent with p'. Note that the algorithm only considers pricings consistent with $\mathcal{S}$. One such pricing that will be eventually considered is R. Let us focus on the corresponding iteration of the loop beginning at line 3, and let $\mathcal{K}$ be the set identified in line 7 of this iteration. Since R is consistent with p', we get by (5) that $v(I, R) \leq p'(I) \leq B(I)/(1+\epsilon)$ for all $I \in \mathcal{J}'[e^*]$, implying that $\mathcal{J}'[e^*] \subseteq \mathcal{K}$. On the other hand, by (5) also we have $v(I, R) \geq p'(I)/(1+\epsilon)$ for all $I \in \mathcal{K} \subseteq \mathcal{I}[e^*]$. Thus

$$v(\mathcal{K}, R) \geq \frac{p'(\mathcal{K})}{1+\epsilon} \geq \frac{p'(\mathcal{J}[e^*])}{1+\epsilon}.$$

Note that the restrictions of p' to $E_L(e^*)$ and $E_R(e^*)$ together with $\mathcal{S} \wedge S(R)$ satisfy the preconditions of (H), with respect to $\mathcal{J}'_L(e^*)$ and $\mathcal{J}'_R(e^*)$, respectively. By induction, the algorithm returns two subsets of intervals $\mathcal{J}_1 \subseteq \mathcal{I}_L(e^*)$ and $\mathcal{J}_2 \subseteq \mathcal{I}_R(e^*)$, and two pricing functions $p_1 : E_L(e^*) \mapsto \mathbb{R}_+$ and $p_2 : E_R(e^*) \mapsto \mathbb{R}_+$, both satisfying with $\mathcal{S} \wedge S(R)$, such that $p_1(I) \leq B(I)$ for all $I \in \mathcal{J}_1$ and $p_2(I) \leq B(I)$ for all $I \in \mathcal{J}_2$, and

$$p_1(\mathcal{J}_1) \geq p'(\mathcal{J}'_L(e^*))/(1+\epsilon) \text{ and } p_2(\mathcal{J}_2) \geq p'(\mathcal{J}'_R(e^*))/(1+\epsilon).$$

Let $\mathcal{J} = \mathcal{K} \cup \mathcal{J}_1 \cup \mathcal{J}_2$ and p be the function defined by (6). Then p satisfies $\mathcal{S} \wedge S(R)$ by construction, and $p(I) \leq B(I)$ for all $I \in \mathcal{J}_1 \cup \mathcal{J}_2$. Moreover, for any $I \in \mathcal{K}$ we have $p(I) \leq (1+\epsilon)v(I,R) \leq B(I)$, by (5) and the definition of $\mathcal{K}$. Thus p and $\mathcal{J}$ satisfy (i), (ii), and moreover,

$$\begin{aligned} p(\mathcal{J}) &= p(\mathcal{K}) + p_1(\mathcal{J}_1) + p_2(\mathcal{J}_2) \geq v(\mathcal{K}, R) + p_1(\mathcal{J}_1) + p_2(\mathcal{J}_2) \\ &\geq \frac{p'(\mathcal{J}'[e^*]) + p'(\mathcal{J}'_L[e^*]) + p(\mathcal{J}'_R[e^*])}{1+\epsilon} = \frac{p'(\mathcal{J}')}{1+\epsilon}. \end{aligned} \tag{7}$$

Finally, we note that any solution $(p, \mathcal{J})$ returned by the algorithm must satisfy (i) and (ii). This is obvious for (i) since only pricings consistent with $\mathcal{S}$ are considered (see line 3). For (ii), this follows by induction for the two sets $\mathcal{J}_1$ and $\mathcal{J}_2$ computed by the algorithm, and by the definition of $\mathcal{K}$ and the fact that p is consistent with R. Thus the solution returned by the algorithm has value at least (7). □

5 The Unsplittable Flow Pricing Problem

In this section we show that there is a quasi-polynomial time approximation scheme for UFP on line graphs, provided that the demands and capacities are integers bounded by a quasi-polynomial in the number of intervals. As a corollary, there is a quasi-polynomial time approximation scheme for the pricing problem on line graphs with limited supply.

We combine the method described in the previous section with the technique of [3]. We assume that the demands and capacities are integers bounded by $L' = 2^{\mathrm{polylog}(m)}$. By rescaling, we can assume, at the loss of a factor of $(1-\epsilon)$ of the optimum, that $B(I) \in [1, nm/\epsilon]$ and $\rho(I) \in [1/L', 1]$, for all $I \in \mathcal{I}$. Let $Q = 1 + \lfloor \log \max_I \{B(I)/\rho(I)\} \rfloor = \mathrm{polylog}(m)$. Given an edge e^* and an ϵ-relative pricing R w.r.t. e^*, we partition the intervals of $\mathcal{I}$ into at most Q classes according to the value of $v(I,R)/\rho(I)$: for $q \in [Q]$, $\mathcal{I}^{q,R} = \{I \in \mathcal{I} : \ 2^{q-1} \leq \frac{v(I,R)}{\rho(I)} \leq 2^q\}$.

We recall the following definition from [3].

Definition 3. (ϵ-Restricted profiles) *Let $e = \{u-1, u\}$ be an edge of E, and $h, \epsilon \in \mathbb{R}_+$, with $1/\epsilon \in \mathbb{Z}_+$. Let $x_1, \ldots, x_{1/\epsilon}$ and $y_1, \ldots, y_{1/\epsilon}$ be start points of edges in E, such that*

$$x_1 \leq x_2 \leq \cdots \leq x_{1/\epsilon} \leq u-1 < u \leq y_{1/\epsilon} \leq \cdots \leq y_2 \leq y_1.$$

Then the vector $(\ell_1, \ldots, \ell_n)$, where

$$\ell_i = \begin{cases} 0, & \text{for } i \leq x_1 \text{ and } i > y_1 \\ j\epsilon h, & \text{for } x_j < i \leq x_{j+1} \text{ and } y_{j+1} < i \leq y_j \\ h, & \text{for } x_{1/\epsilon} < i \leq y_{1/\epsilon}, \end{cases}$$

is said to be an ϵ-restricted profile *with peak e and* height *h, denoted by $RP_\epsilon(e; h; x_1, \ldots, x_{1/\epsilon}; y_1, \ldots, y_{1/\epsilon})$.*

The total number of ϵ-restricted profiles, with a given peak and height is at most $n^{2/\epsilon}$.

For $\mathcal{J} \subseteq \mathcal{I}$ and any edge $e \in E$, the total demands of $\mathcal{J}[e]$ define a profile $\text{prof}(\mathcal{J}[e]) \in \mathbb{R}^E$, defined by:

$$\text{prof}(\mathcal{J}[e])_{e'} = \rho(\mathcal{J}[e] \cap \mathcal{J}[e']) \stackrel{\text{def}}{=} \sum_{I \in \mathcal{J}[e] \cap \mathcal{J}[e']} \rho(I), \quad \text{for } e' \in E.$$

The following two lemmas state that any such profile can be sufficiently accurately approximated by an ϵ-restricted profile, in polynomial time (see Figure 3), provided that the demands are sufficiently small.

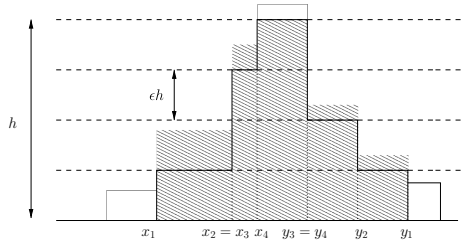


Fig. 3. A profile and its restriction

Lemma 4 ([3]). *Let $e^* \in E$ be an edge, R be an ϵ-relative pricing w.r.t. e^*, and $\mathcal{J} \subseteq \mathcal{I}^{q,R}[e^*]$ be a subset of class q intervals spanning e^*, such that $\rho(I) \leq \delta$ for all $I \in \mathcal{J}$ and some $\delta \in \mathbb{R}_+$. Let h be the largest integer multiple of $1/2^q$ that does not exceed $\rho(\mathcal{J})$. Then there exists an ϵ-restricted profile π with peak e^* and height h and a subset $\mathcal{J}' \subseteq \mathcal{J}$, such that*

(i) $\text{prof}(\mathcal{J}') \leq \pi \leq \text{prof}(\mathcal{J})$, *and*
(ii) $v(\mathcal{J}, R) - v(\mathcal{J}', R) \leq 2^{q+1}(\epsilon h + \delta)$.

Lemma 5 ([3]). *Let e^* be an edge, R an ϵ-relative pricing w.r.t. e^*, and $\mathcal{J} \subseteq \mathcal{I}^{q,R}[e^*]$ be a subset of class q intervals such that $\rho(I) \leq \delta$ for all $I \in \mathcal{J}$. Let π be a restricted profile, and $\mathcal{J}^*$ be a subset of $\mathcal{J}$ such that* $\text{prof}(\mathcal{J}^*) \leq \pi$ *and $v(\mathcal{J}^*, R)$ is maximized. Then we can find in polynomial time a subset $\mathcal{J}' \subseteq \mathcal{J}$ such that* $\text{prof}(\mathcal{J}') \leq \pi$ *and $v(\mathcal{J}^*, R) - v(\mathcal{J}', R) \leq 2^{q+1}\delta/\epsilon$.*

Remark 1. If $\rho(I) = 1$ for all $I \in \mathcal{J}$, then a subset satisfying the conditions in Lemma 5 with $v(\mathcal{J}', R) = v(\mathcal{J}^*, R)$ can be found by a greedy procedure as follows. Consider the intervals in $\mathcal{J}$ in increasing order of their $v(\cdot, R)$ values, and let $\mathcal{J}'$ be a maximal set of intervals such that $\text{prof}(\mathcal{J}') \leq \pi$. Then it is easy to see that $v(\mathcal{J}', R) \geq v(\mathcal{J}^*, R)$.

The algorithm is an extension of HP and is given in Figure 4 below. It resembles very much the algorithm in [3], with the additional incorporation of ϵ-relative pricings. Without loss of generality we can assume that $c(e) < \infty$ for all $e \in E$

Algorithm UFP$(\mathcal{I}, E, \mathcal{S}, c)$**:**
Input: A subset of intervals $\mathcal{I}$ defined on E, a feasible system of linear inequalities $\mathcal{S}$, and a capacity vector $c = (c(e) : e \in E)$
Output: A price function $p : E \mapsto \mathbb{R}_+$ and a subset $\mathcal{J} \subseteq \mathbb{R}_+$ s.t. $p(I) \leq B(I)\ \forall I \in \mathcal{J}$ and $\rho(\mathcal{J}[e]) \leq c(e)\ \forall e \in E$

1. **if** $|\mathcal{I}| = 0$, **then return** $(p, \emptyset)$, where p is any feasible solution of $\mathcal{S}$
2. let e^* be an edge of E such that $|\mathcal{I}_L(e^*)| \leq m/2$ and $|\mathcal{I}_R(e^*)| \leq m/2$
3. **for** every ϵ-relative pricing R w.r.t. e^* for which $\mathcal{S} \wedge S(R)$ is feasible **do**
4. **foreach** $q \in [Q]$, set $\mathcal{I}_q = \{I \in \mathcal{I}^{q,R}[e^*] :\ v(I,R) \leq B(I)/(1+\epsilon)\}$
5. **foreach** Q-tuple $(\mathcal{A}_1, \ldots, \mathcal{A}_Q)$ with each $\mathcal{A}_q \subseteq \mathcal{I}_q$ and $|\mathcal{A}_q| \leq 1/\epsilon^2$ **do**
6. $\mathcal{A} \leftarrow \bigcup_{q=1}^{Q} \mathcal{A}_q$
7. **if** $\rho(\mathcal{A}) \leq c(e)\ \forall e \in E$ **then**
8. **foreach** $e \in E$, set $c'(e) \leftarrow c(e) - \rho(\mathcal{A}[e])$
9. **foreach** $(h_1, \ldots, h_Q) \in \mathbb{R}_+^Q$ with each h_q an integer multiple of $1/2^q$ and $\sum_{q=1}^{Q} h_q \leq c'(e)$ **do**
10. **foreach** Q-tuple $(\pi_1, \ldots, \pi_Q)$ with each π_q an ϵ-restricted profile with peak e^* and height h_q, s.t. $\sum_{q=1}^{Q} \pi_q \leq c'$ **do**
11. **foreach** $q \in [Q]$, set $\mathcal{B}_q \leftarrow \{I \in \mathcal{I}_q \setminus \mathcal{A}_q :\ \rho(I) \leq \epsilon^2(h_q + 1/2^q + \rho(\mathcal{A}_q))\}$
12. **foreach** $q \in [Q]$, let $\mathcal{K}_q \leftarrow$PACK$(\mathcal{B}_q, \pi_q, R)$
13. $\mathcal{K} \leftarrow \bigcup_{q=1}^{Q} \mathcal{K}_q$
14. **foreach** $e \in E$, set $c''(e) \leftarrow c'(e) - \rho(\mathcal{B}[e])$
15. $(p_1, \mathcal{J}_1) \leftarrow$ UFP$(\mathcal{I}_L(e^*), E_L(e^*), \mathcal{S} \wedge S(R), c'')$
16. $(p_2, \mathcal{J}_2) \leftarrow$ UFP$(\mathcal{I}_R(e^*), E_R(e^*), \mathcal{S} \wedge S(R), c'')$
17. let p be the price function defined by (6)
18. $\mathcal{J} \leftarrow \mathcal{A} \cup \mathcal{K} \cup \mathcal{J}_1 \cup \mathcal{J}_2$
19. record $(p, \mathcal{J})$
20. **return** the recorded solution with largest $p(\mathcal{J})$ value

Fig. 4. The dynamic program for computing ϵ-approximate prices for the capacitated version

by setting $c_e \leftarrow \min\{c_e, \rho(\mathcal{I}[e])\}$. For a subset of intervals $\mathcal{J}$, a restricted profile π, and a relative pricing R, we use PACK$(\mathcal{J}, \pi, R)$ to denote a procedure that returns a subset with the guarantees of Lemma 5.

The algorithm proceeds as before, by considering the middle edge e^*. It partitions the set of candidate intervals crossing e^* in a given class $q \in [Q]$ into those with large demands $\mathcal{A}_q$, and those with small demands $\mathcal{B}_q$. The number of large demands crossing e^* is small (less than $1/\epsilon^2$) and hence they can be guessed by enumerating over all of them. For each such guess of large demands, the profile of the small demands crossing e^* can be approximated by an ϵ-restricted profile using Lemmas 4 and 5. Again, among these profiles, we enumerate over the ones whose total demand requirement does not exceed the residual capacity left after routing the large demands (see [3] for more details).

Lemma 6. *Algorithm* UFP *runs in quasi-polynomial time in m, for any fixed $\epsilon > 0$.*

Lemma 7. *Algorithm* UFP *returns a solution of value at least* $(1-23\epsilon)p^*(\text{OPT})$*, for any* $0 < \epsilon < 1$.

Theorem 2. *There is a quasi-polynomial time approximation scheme for* UFP *on line graphs, provided that the demands and capacities are integers bounded by a quasi-polynomial in the number of intervals.*

Corollary 1. *There is a quasi-polynomial time approximation scheme for the pricing problem on line graphs with limited supply.*

References

1. Aggarwal, G., Hartline, J.D.: Knapsack auctions. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 1083–1092. ACM Press, New York, USA (2006)
2. Balcan, M.F., Blum, A.: Approximation algorithms and online mechanisms for item pricing. In: EC '06: Proceedings of the 7th ACM conference on Electronic commerce, pp. 29–35. ACM Press, New York, USA (2006)
3. Bansal, N., Chakrabarti, A., Epstein, A., Schieber, B.: A quasi-PTAS for unsplittable flow on line graphs. In: STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing, pp. 721–729. ACM Press, New York, USA (2006)
4. Briest, P., Krysta, P.: Single-minded unlimited supply pricing on sparse instances. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 1093–1102. ACM Press, New York, USA (2006)
5. Briest, P., Krysta.: Buying cheap is expensive: Hardness of non-parametric multi-product pricing, Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM-SIAM (2007)
6. Demaine, E.D., Hajiaghayi, M.T., Feige, U., Salavatipour, M.R.: Combination can be hard: approximability of the unique coverage problem. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 162–171. ACM Press, New York, USA (2006)
7. Glynn, P.W., Van Roy, B., Rusmevichientong, P.: A nonparametric approach to multi-product pricing. Operations Research 54
8. Guruswami, V., Hartline, J.D., Karlin, A.R., Kempe, D., Kenyon, C., McSherry, F.: On profit-maximizing envy-free pricing, SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, pp. 1164–1173 (2005)
9. Hartline, J.D., Koltun, V.: Near-optimal pricing in near-linear time. In: Dehne, F., López-Ortiz, A., Sack, J.-R. (eds.) WADS 2005. LNCS, vol. 3608, pp. 422–431. Springer, Heidelberg (2005)

Improved Upper Bounds on the Competitive Ratio for Online Realtime Scheduling

Koji Kobayashi[1] and Kazuya Okamoto[2]

[1] Graduate School of Informatics, Kyoto University
kobaya@net.ist.i.kyoto-u.ac.jp
[2] Graduate School of Informatics, Kyoto University
okia@kuis.kyoto-u.ac.jp

Abstract. We study a variant of online scheduling problems, the online realtime scheduling. It can be defined on a complete graph, where each node represents a communication agent, and a communication between two agents can be considered as an edge. An input is a sequence of communication jobs, each of which requires two specified agents to communicate during specified time period. Each agent can participate in at most one communication job. The task of an online algorithm is to schedule jobs so that the sum of the profits of completed communication jobs is maximized. In this paper, we improve the competitive ratio of the General Shelf based Max Matching ($GSMM$) algorithm from $6 + 4\sqrt{2}(\approx 11.66)$ to $2\sqrt{6} + 6(\approx 10.90)$. We also prove that this ratio is optimal for $GSMM$. In addition, we study the case where each job has no slack time, namely, it must be either started immediately or rejected at its release time, and show the competitive ratio of $GSMM$ is $2\sqrt{6} + 5(\approx 9.90)$.

1 Introduction

In parallel and distributed environments, the following communication scheduling problem sometimes arises. There are agents, with communication lines between pairs of agents. To perform some task, two agents must communicate for some time span, which is called *a communication job*. Each communication job has its own profit and its own deadline. Such communication jobs arrive online, which may require one agent to serve two or more jobs simultaneously. However, each agent can participate in at most one job at the same time. Our goal is to schedule jobs so that the sum of the profits of completed jobs is maximized.

Lee et al.[9] formulated this problem as an online problem on a graph, called *online realtime scheduling*, in the following way: We have a graph where each node is considered as an agent, and each edge represents a communication line between two associated agents. An input is a sequence of communication jobs. Each communication job J is specified by an edge $e = (v_1, v_2)$, a release time r, a profit v, a processing time p (also, called length), and a deadline d. This means that this job requires a communication between agents v_1 and v_2, time needed for communication is p, and the communication should be performed between time r and d. For a job to be completed, it must be assigned to an edge during

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 463–474, 2007.

time p without interruption. A release time is usually the same to the time when the job is given. The task of an online algorithm is to schedule the assignment of jobs to edges so that at any time, each node receives at most one job. A job is called *accepted* if the communication is completed. When a job, say J_1, arrives at an edge whose endpoint is already occupied by another job, say J_2, an algorithm can preempt J_2 and assign J_1. It can later re-schedule J_2, but in that case, J_2 must be assigned to its destination edge for the processing time required for J_2, namely, the task already performed for J_2 before J_1 arrives would be aborted. $d-r-p$ denotes the *slack time* of J, namely, if this job is not started within this time period, then it cannot be completed by the deadline. If a job has no slack time, an algorithm must decide to schedule or reject it immediately. The profit of an algorithm is the sum of the profits of all accepted jobs. The performance of an online algorithm is evaluated by the competitive analysis [2,11]. If, for any input σ, the profit of an online algorithm ALG is at least $1/c$ of that of an optimal offline algorithm for σ, then we say that ALG is c-competitive.

Lee et al. [9] have considered this problem on a complete graph. They considered a restricted case where the profit is equal to the length for any communication job. They showed that for unit jobs case (i.e., length of any job is 1) with any slack time, a simple greedy algorithm is 2-competitive [9], and no online algorithm can be better than 1.5-competitive. They also considered the more general case such that parameters can take any real values. They showed that for any jobs with any slack time, the General Shelf based Max Matching ($GSMM$) algorithm is $(6+4\sqrt{2} \approx 11.66)$-competitive and no online algorithm can have competitive ratio better than $8-\epsilon$ for any positive constant ϵ. This can be applied to the case where each job has no slack time.

Our Contribution. In this paper, we give detailed analysis for $GSMM$, and improve the upper bound on the competitive ratio of $GSMM$ from $6+4\sqrt{2}(\approx 11.66)$ to $2\sqrt{6}+6(\approx 10.90)$. We also give a matching lower bound example showing that the competitive ratio of $GSMM$ is no better than $2\sqrt{6}+6$. For the case where each arriving job has no slack time, we show the competitive ratio of $GSMM$ is $2\sqrt{6}+5(\approx 9.90)$.

Let us briefly explain an idea of the analysis. Let EX be the set of jobs which optimal offline algorithm (OPT) accepts but $GSMM$ does not. We want to bound the sum of the values of jobs in EX. For this purpose, we will show that when OPT starts any job $I \in EX$, $GSMM$ schedules some job J instead, and the value of job I can be bounded using the value of job J.

Related Results. Online realtime scheduling treated in [9] and this paper is defined on a complete graph, but in general, it can be considered on arbitrary graphs. Guez et al. [6] consider the problem on a bipartite graph. In this model, one node set denotes the senders, the other set the receivers, and a simultaneous transmission of messages is a bipartite matching. The conflict constraint, namely, the restriction that each node can serve at most one job simultaneously, was first introduced in [7].

The problem where jobs arrive at not edges but nodes is the most basic realtime scheduling problem, and a great amount of study have been done up to the present (e.g., Baruah et al. [1], Woeginger [12], and Koren and Shasha [8]). There are a lot of variants of this problem, according to the availability of preemption, existence of slack time, and the profit of each job. (We say that a job J has no slack time if J can be accepted only when J is started at its release time.) In the case where a value of each job is equal to the length of a job, preemption is allowed, and there is no slack time, Baruah et al. [1] and Woeginger [12] showed that no online algorithm can be better than 4-competitive, and Baruah et al. [1] gave a 4-competitive online algorithm [1]. If the value and the length of job may be different, an upper bound $(\sqrt{k}+1)^2$ is presented [1], where k is the ratio between the maximum and the minimum length of jobs. Later, Koren and Shasha gave a lower bound which matches this upper bound [8]. [3,4,5] study the model where preemption is not allowed and each job can have arbitrary slack time. Existence of slack time helps not only an online algorithm but also an optimal one [5].

2 Preliminaries

In this section, we formally define a problem studied in this paper, and General Shelf based Max Matching ($GSMM$) algorithm introduced in [9].

2.1 Online Realtime Scheduling

We have a complete graph $G = (V, E)$. A job J is specified by a 3-tuple $J = (r, p, d)$, where r is the release time, p is the processing time (length) and d is the deadline. An input is a sequence of jobs, each of which arrives (or occurs) at an edge $e \in E$. The task of an online algorithm is to schedule jobs at any time, so that no endpoint node participate in more than one jobs, namely, at any time, a set of scheduled jobs forms a matching in G. A job J is *accepted* if it is scheduled to its destination edge without interruption for time p between its release time r and deadline d; otherwise it is *rejected*. We suppose that a job J_1 is already assigned to an edge, say $e_1 = (v_1, v_2)$. When another J_2 arrives at an edge $e_2 = (v_1, v_3)$, an algorithm can assign J_2 to e_2 by preempting J_1. In that case, we say that J_1 is *preempted*. J_1 can be later assigned to e_1 and can be accepted, but in that case, J_1 must be assigned to e_1 during the length of J_1, which means that the task already performed by e_1 before J_2 arrives is aborted. Let $s = d - r - p$ be the *slack time* of J. We study the case where each job has slack time in Sec. 3 and Sec. 4, and no slack time in Sec. 5. The *value* an algorithm ALG obtains for an input σ, denoted by $V_{ALG}(\sigma)$, is the sum of the lengths of jobs accepted by ALG. If $V_A(\sigma) \geq V_{OPT}(\sigma)/c$ for an arbitrary input σ, we say that A is *c-competitive*, where OPT is an optimal offline algorithm for σ. Without loss of generality, we can assume that OPT never preempts jobs.

2.2 General Shelf Based Max Matching ($GSMM$) Algorithm

We give some notations. For a job J, let $p(J)$, $d(J)$ and $r(J)$ denote the length, the deadline and the release time of J, respectively. For a set of jobs K, define $p(K) = \sum_{j \in K} p(j)$, namely, the sum of the lengths of all jobs in K. A job J is *available* at time t if it can be started at time t and accepted in the future, i.e. $d(J) \geq t + p(J)$ and $r(J) \leq t$ hold.

$GSMM$ is defined as follows: It executes computation at time t when a new job arrives or some job is accepted. $GSMM$ takes a positive number $k(\geq 2)$ as a parameter. M denotes the set of scheduled jobs (namely, those jobs assigned to its destination edge and currently executed jobs) at time t, e.g., M is empty at the beginning.

General Shelf based Max Matching ($GSMM(k)$)

Step 1: Let $\mathcal{S}$ be the set of jobs which have already arrived at time t, namely, $\mathcal{S} = \{J | r(J) \leq t\}$.

Step 2: Remove jobs accepted before t from $\mathcal{S}$.

Step 3: Remove unavailable jobs from $\mathcal{S}$.

Step 4: Give each job J a weight $w_t(J)$ at time t in the following way:
If $J \in M$, $w_t(J) = kp(J)$, and otherwise $w_t(J) = p(J)$.

Step 5: Constitute a weighted (multi)graph $G' = (V', E')$ as follows:
V' is V, the same to the nodes of G, and E' is the set of jobs in $\mathcal{S}$.
The weight of $J \in E'$ is $w_t(J)$.
Compute a maximum matching X in G', and schedule jobs in X, namely, set $M = X$.

As one can see, $GSMM$ tries to schedule jobs whose profits are large. However, it gives weights, parameterized by k, to currently scheduled jobs.

3 $2\sqrt{6} + 6$ Upper Bound

For analysis, we give some definitions. Recall that our problem is defined on a complete graph $G = (V, E)$. We assume that nodes in V are specified by numbers from 1 to $|V|$. For a job J, $e(J) \in E$ denotes the edge which J arrives at. Let $e(J) = (u, v)$ such that $u < v$. Then we write $v_1(J) = u$ and $v_2(J) = v$, namely, $v_1(J)$ and $v_2(J)$ represent the smaller and the larger nodes of the edge J arrives at, respectively. A job J *conflicts* with a job I at time t if both of them are available at t and the edges J and I destined share a common endpoint, namely, $v_j(J) = v_i(I)$ for $\exists i, \exists j \in \{1, 2\}$. In particular, a job conflicts with itself. A job J *j-conflicts* ($j = 1, 2$) with I at time t if J conflicts with I at t and $v_j(J) = v_i(I)$ for some i.

3.1 Overview of the Analysis

If EX denotes the set of jobs which OPT accepts but $GSMM(k)$ rejects, $V_{OPT}(\sigma) \leq V_{GSMM(k)}(\sigma) + p(EX)$ for any input σ. In the following subsections, we evaluate the upper bound of $p(EX)$ to estimate the competitive ratio of $GSMM(k)$. We show below the outline of the proof.

Let us classify jobs started by $GSMM(k)$ at time t into two categories: Let A_t be the set of jobs which are scheduled between $[t, t + p(J))$ and hence accepted. Let P_t be the set of jobs preempted at time $t' \in (t, t+p(J))$. Now, let $\mathcal{T}$ be the set of times when $GSMM(k)$ newly starts a job. Then, $V_{GSMM(k)}(\sigma) = \sum_{t\in\mathcal{T}} p(A_t)$ holds.

In Sec. 3.2, we show that $p(EX) \leq (2k+2)\sum_{t\in\mathcal{T}} p(A_t) + (k+2)\sum_{t\in\mathcal{T}} p(P_t)$, and in Sec. 3.3, we prove that $\sum_{t\in\mathcal{T}} p(P_t) \leq \frac{1}{k-1}\sum_{t\in\mathcal{T}} p(A_t)$ holds. Therefore, $V_{OPT}(\sigma) \leq p(EX) + V_{GSMM(k)}(\sigma) \leq (2k + 2 + \frac{k+2}{k-1} + 1)V_{GSMM(k)}(\sigma)$, which implies that the competitive ratio of $GSMM(k)$ is at most $2k+2+\frac{k+2}{k-1}+1$. We have that $2k + 2 + \frac{k+2}{k-1} + 1 = 2\sqrt{6} + 6$ when $k = 1 + \sqrt{6}/2$. Hence, we have the following theorem:

Theorem 3.1. *The competitive ratio of* $GSMM(1+\sqrt{6}/2)$ *is at most* $2\sqrt{6}+6$.

3.2 Evaluating $p(EX)$ with $p(A_t)$ and $p(P_t)$

In this section, for a job $I \in EX$ started at time t by OPT, we will show that $GSMM(k)$ surely schedules a job which conflicts with I at time t.

Lemma 3.2. *Let I be a job in EX, and suppose that OPT starts I at time t. Then, there exists a job J which $GSMM(k)$ schedules at time t and conflicts with I at time t.*

Proof. Recall that OPT never preempts a job. So, I is eventually accepted by OPT. Hence, I is available at time t. Also, observe that by the definition of $GSMM$, the set of scheduled jobs by $GSMM$ constitutes a maximal matching in the graph defined by all available jobs. However, if there is no job which $GSMM$ schedules and conflicting with I at t, it contradicts the maximality. This completes the proof.

We give some more definitions. Let $I \in EX$ and suppose that OPT starts I at time t. Using Lemma 3.2, we can guarantee the existence of jobs being scheduled by $GSMM(k)$ at t and conflicting with I at t. The number of such jobs may 1 or 2. (1) If there is only one such job, say J, we say that J *covers* the whole length (namely, $p(I)$) of I. (2) If there are two such jobs, say J and J', then we say that J *covers* $\frac{p(J)}{p(J)+p(J')}$-fraction of the length of I in chronological order, namely, $p(I)\frac{p(J)}{p(J)+p(J')}$. Note that in this case J' covers $p(I)\frac{p(J')}{p(J)+p(J')}$ of I, and the sum of the lengths J and J' cover is equal to $p(I)$, the whole length of I. For a job J scheduled by $GSMM(k)$, let $C(J)$ denotes the sum of the lengths of jobs which J covers (note that J may cover more than one jobs). For a set of jobs K, let $C(K) = \sum_{J\in K} C(J)$.

Lemma 3.3

$$p(EX) = \sum_{t \in \mathcal{T}} C(A_t) + \sum_{t \in \mathcal{T}} C(P_t).$$

Proof. Let I be an arbitrary job in EX. By definition, I is certainly covered by one or two jobs scheduled by $GSMM$. Therefore, we have the lemma.

In order to evaluate the length which a job $J \in P_t \cup A_t$ covers, we will consider the property of the job started by $GSMM(k)$ in place of a preempted job.

Lemma 3.4. *Let $J \in P_t$ be a job scheduled by $GSMM(k)$ between t and t'. Then, there exists a job $\tilde{J}$ which is started by $GSMM(k)$ and conflicts with J at t' such that $p(J) \leq p(\tilde{J})$.*

Proof. Note that t' is the time when $GSMM(k)$ preempts J. Since $J \in P_t$, $GSMM(k)$ starts another job J' by preempting J. We consider three cases according to the nodes at which J conflicts with J' at t':

Case 1. J 1-conflicts and 2-conflicts with J'
In this case, $GSMM(k)$ preempts J and newly schedules J' at the same edge. So, by the definition of $GSMM(k)$, $kp(J) \leq p(J')$ and hence J' is the job which we want as $\tilde{J}$.

Case 2. J only 1-conflicts with J'
If $p(J) \leq p(J')$, then J' satisfies our condition. Otherwise, there exists J'' which is 2-conflicted with J such that $kp(J) \leq p(J') + p(J'')$ at time t'. By this inequality and the fact that $p(J) > p(J')$, we have that $(k-1)p(J) < p(J'')$. By this inequality and the assumption that $k \geq 2$, $p(J) < p(J'')$. Hence J'' is $\tilde{J}$ which we want to show the existence.

Case 3. J only 2-conflicts with J'
Similar to the proof of Case 2, and hence omitted.

For any $J \in P_t$ scheduled by $GSMM(k)$ between t and t', there exists a job $\tilde{J}$ started by $GSMM(k)$ and conflicting with J at t' such that $p(J) \leq p(\tilde{J})$ by Lemma 3.4. We denote this $\tilde{J}$ by $N(J,t)$.

In the following, we use the following convenience. Let $I \in EX$ be a job accepted by OPT, and suppose that it is scheduled between t_1 and t_2 by OPT and covered by a job $J \in P_t$. We may consider *dividing I into I_1 and I_2 at time $t' \in [t_1, t_2)$*, so that OPT accepts both I_1 and I_2 instead of I and schedules I_1 and I_2 between $[t_1, t')$ and $[t', t_2)$, respectively. Then, when a job $N(J,t)$is started at time t', we divide the fraction of length of a job I covered by J into I_1 and I_2 at t'. Then, we I_2 denote $S(J,t,I)$.

Lemma 3.5. *Let $\tilde{J}$ be a job $N(J,t)$. We suppose that the fraction of length of a job I (say, I') is covered by J and I' is divided into I_1 and $I_2 = S(J,t,I)$ at t'. Then, the length of $S(J,t,I)$ is at most $kp(\tilde{J})$.*

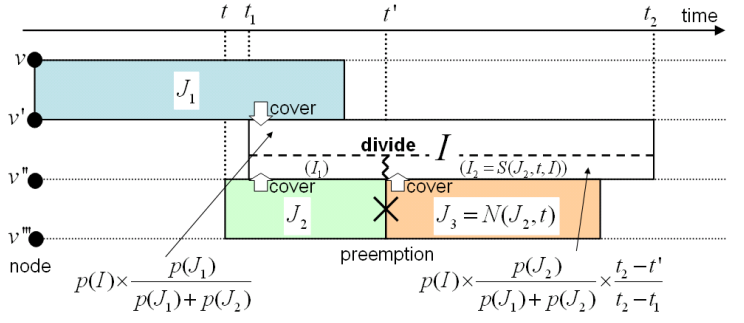


Fig. 1. Example of dividing

Proof. By Lemma 3.4, $p(J) \leq p(\tilde{J})$. Since $GSMM(k)$ does not preempt J and schedule I, $p(I') \leq kp(J)$. Therefore, we have that $p(I_2) < p(I') \leq kp(J) \leq kp(\tilde{J})$.

By Lemma 3.5, the $S(J,t,I)$ from any job $I \in EX$ and the job $J \in P_t$ which covers the fraction of the length of I can be dealt similarly to other jobs in EX. Next, we evaluate the upper bound of $C(J)$ for a job $J \in A_t$, namely, a job J accepted by $GSMM(k)$.

Lemma 3.6. *For any t and for any J such that $J \in A_t$, $C(J) \leq (2k+2)p(J)$.*

Proof. Assume that J is scheduled by $GSMM(k)$ between t_0 and t_1. We estimate the total length of jobs covered by J. Let $J_1, J_2, \ldots, J_\ell$ be jobs covered by J that 1-conflict with J in this order. Then the total length J covers these jobs is $\sum_{j=1}^{\ell} p(J_j)$. We consider two cases (i) J_ℓ is accepted by OPT at or before t_1, and (ii) J_ℓ is accepted by OPT after t_1. In case (i), J_1 is scheduled at or after t_0, and J_ℓ ends before t_1. Hence $\sum_{j=1}^{\ell} p(J_j) \leq t_1 - t_0 = p(J)$. In case (ii), we can show that $\sum_{j=1}^{\ell-1} p(J_j) \leq p(J)$ by the same argument. We then show that $p(J_\ell) \leq kp(J)$. First, consider the case that J_ℓ is covered by only J. Then, clearly $p(J_\ell) \leq kp(J)$ since otherwise, $GSMM$ must have scheduled J_ℓ instead of J. Next, consider the case that J_ℓ is covered by two jobs J and J'. Then $p(J_\ell) \leq k(p(J) + p(J'))$ since otherwise, $GSMM$ must have scheduled J_ℓ instead of J and J'. So, the length J covers J_ℓ is $p(J_\ell)\frac{p(J)}{p(J)+p(J')} \leq k(p(J) + p(J'))\frac{p(J)}{p(J)+p(J')} = kp(J)$. So, in case (ii), $\sum_{j=1}^{\ell} p(J_j) \leq (k+1)p(J)$. Now, we have proved that in either (i) and (ii), $\sum_{j=1}^{\ell} p(J_j) \leq (k+1)p(J)$.

We can do a similar argument to show that the sum of the covered length of jobs that 2-conflict with J is at most $(k+1)p(J)$. Hence, the total length covered by J is at most $2(k+1)p(J)$.

Then, we evaluate the upper bound of $C(J)$ for a job $J \in P_t$, namely, a job J preempted by $GSMM(k)$.

Lemma 3.7. *For any t and for any $J \in P_t$, $C(J) \leq (k+2)p(J)$.*

Proof. Let $J \in P_{t_0}$ be a job scheduled by $GSMM(k)$ between $[t_0, t_1)$. By Lemma 3.4, there exists a job J' started by $GSMM(k)$ at time t_1 conflicted with J such that $p(J') \geq p(J)$. Without loss of generality, we assume that J 1-conflicts with J' at time t_1.

We consider the following two cases: (1) There exists $I \in EX$ such that I is started between t_0 and t_1 by OPT, and accepted after t_1, and 1-conflicting with J, and (2) otherwise.

We first consider the easier case, Case (2). Since we can consider similarly to the proof of Lemma 3.6, the total length J covers jobs accepted by OPT at or before t_1 is at most $2p(J)$ and the length J covers a job accepted by OPT is at most $kp(J)$. since we think only of a node $v_2(J)$. Hence, $C(J) \leq (k+2)p(J)$.

Next, we consider the case (1). We divide I into I_1 and I_2 at time t_1. Hence, J covers the length of I_1 scheduled between $[t'_0, t_1)$ and J' covers the length of I_2 scheduled between $[t_1, t'_1)$. Since the total length J covers jobs accepted by OPT at or before t_1 is at most $2p(J)$ and the length J covers a job accepted by OPT is at most $kp(J)$ since we think only of $v_2(J)$. Therefore, we have that $C(J) \leq (k+2)p(J)$ holds again, which completes the proof.

The following corollaries are immediate from Lemmas 3.6 and 3.7.

Corollary 3.8. $C(A_t) \leq (2k+2)p(A_t)$.

Corollary 3.9. $C(P_t) \leq (k+2)p(P_t)$.

We have finally the following lemma:

Lemma 3.10. $p(EX) \leq (2k+2)\sum_{t\in\mathcal{T}} p(A_t) + (k+2)\sum_{t\in\mathcal{T}} p(P_t)$.

Proof. Immediate from Lemma 3.3, Corollary 3.8 and Corollary 3.9.

3.3 Bounding $p(P_t)$ Using $p(A_t)$

Finally, in this section, we bound $\sum_{t\in\mathcal{T}} p(P_t)$ using $\sum_{t\in\mathcal{T}} p(A_t)$. Let P'_t denote the set of jobs which are scheduled by $GSMM(k)$ and preempted at time t.

Lemma 3.11. $\sum_{t\in\mathcal{T}} p(P_t) \leq \frac{1}{k-1}\sum_{t\in\mathcal{T}} p(A_t)$.

Proof. First, note that for any time t, $kp(P'_t) \leq p(P_t) + p(A_t)$ since $GSMM(k)$ newly starts jobs in $P_t \cup A_t$ by preempting jobs in P'_t. By summing up this inequality for all $t \in \mathcal{T}$, we have

$$\sum_{t\in\mathcal{T}} kp(P'_t) \leq \sum_{t\in\mathcal{T}} p(P_t) + \sum_{t\in\mathcal{T}} p(A_t).$$

Now, for any job J preempted by $GSMM(k)$, suppose that $GSMM(k)$ starts J at t_1 and preempts it at t_2. Then, J contributes to both P_{t_1} and P'_{t_2}, and

$t_2 \in \mathcal{T}$ since at least one job is started whenever a job is preempted. Thus, $\sum_{t\in\mathcal{T}} p(P'_t) = \sum_{t\in\mathcal{T}} p(P_t)$, and hence we have that

$$\sum_{t\in\mathcal{T}} p(P_t) \le \frac{1}{k-1}\sum_{t\in\mathcal{T}} p(A_t),$$

which completes the proof.

4 Tight Lower Bound of $GSMM(k)$

In this section, we prove that the competitive ratio of $GSMM(k)$ is not better than $2\sqrt{6}+6$ for any k, which matches the upper bound given by the previous section.

Theorem 4.1. *For any positive constant ε, the competitive ratio of $GSMM(k)$ is at least $2\sqrt{6}+6-\varepsilon$.*

Proof. Let F be an arbitrary positive integer. We consider the complete graph with $3F+5$ vertices. The input σ, giving a lower bound example, consists of $4F+5$ jobs which is summarized in the following table.

Jobs J_j, J'_j, J''_j, J'''_j $(j=0,1,\ldots,F)$ and $\tilde{J}$ arrive according to the following table, where σ'_j, σ''_j, σ'''_j, $\tilde{\sigma}$, ϵ_j, ϵ'_j, ϵ''_j and ϵ'''_j are sufficiently small constants such that $\sigma'''_j > 0$, $\sigma''_j > \sigma'_j > 0$, $\epsilon''_j > \epsilon'''_j > \epsilon'_j > \epsilon_j > 0$, $\tilde{\sigma} > 0$ and $\sum_{j=0}^{F}(\sigma'_j + \sigma''_j + \sigma'''_j + \epsilon'_j + \epsilon''_j) + \tilde{\sigma} = \varepsilon p(J_F)$ for any j. Finally, let L be a positive value such that $L > \max_j\{\frac{\sigma''_j+\epsilon''_j}{k^j}\}$ and $L > \max_j\{\frac{\sigma'''_j}{k^{j+1}}\}$

arriving jobs	release time	length	smaller node	larger node	slack time
J_0	0	L	1	2	0
J_{j+1} $(j=0,\ldots,F-1)$	$r(J_j)$ $+p(J_j)-\epsilon_j$	$kp(J_j)$	1	$j+3$	0 $(j=0,\ldots,F-2)$ ∞ $(j=F-1)$
$\tilde{J}$	$r(J_F)$ $+p(J_F)-\epsilon_F$	$kp(J_F)-\tilde{\sigma}$	1	$3F+5$	0
J'_j $(j=0,\ldots,F)$	$r(J_j)+\sigma'_j$	$p(J_j)$ $-\sigma'_j-\epsilon'_j$	1	$F+j+3$	0
J''_j $(j=0,\ldots,F)$	$r(J_j)+\sigma''_j$	$p(J_j)$ $-\sigma''_j-\epsilon''_j$	$j+2$	$2F$ +j+4	0
J'''_j $(j=0,\ldots,F)$	$r(J_j)$ $+p(J_j)-\epsilon'''_j$	$kp(J_j)-\sigma'''_j$	$j+2$	$2F$ $+j+4$	0

Then, by the definition of $GSMM(k)$, it cannot start J'_j, J''_j, J'''_j and $\tilde{J}$, and preempts J_{j-1} in starting J_j. Hence, $GSMM(k)$ accepts only J_F. On the other hand, OPT accepts J'_j, J''_j,J'''_j for all j and $\tilde{J}$. Hence, we have that

$$\begin{aligned}
\frac{V_{OPT}(\sigma)}{V_{GSMM(k)}(\sigma)} &= \left(kp(J_F) + p(J_F) + \sum_{j=0}^{F}(2+k)p(J_j) - \varepsilon p(J_F) \right) \frac{1}{p(J_F)} \\
&= \left(kp(J_F) + p(J_F) + \sum_{j=0}^{F}(2+k)\frac{p(J_F)}{k^{F-j}} - \varepsilon p(J_F) \right) \frac{1}{p(J_F)} \\
&= \left((k+1)p(J_F) + \sum_{j=0}^{F}(2+k)\frac{p(J_F)}{k^{F-j}} - \varepsilon p(J_F) \right) \frac{1}{p(J_F)} \\
&= k+1+(k+2)\frac{k-\frac{1}{k^F}}{k-1} - \varepsilon \overset{F\to\infty}{\longrightarrow} k+1+(k+2)\frac{k}{k-1} - \varepsilon.
\end{aligned}$$

Therefore, the competitive ratio of $GSMM(k)$ is at least $k+1+(k+2)\frac{k}{k-1}-\varepsilon \geq 2\sqrt{6}+6-\varepsilon$.

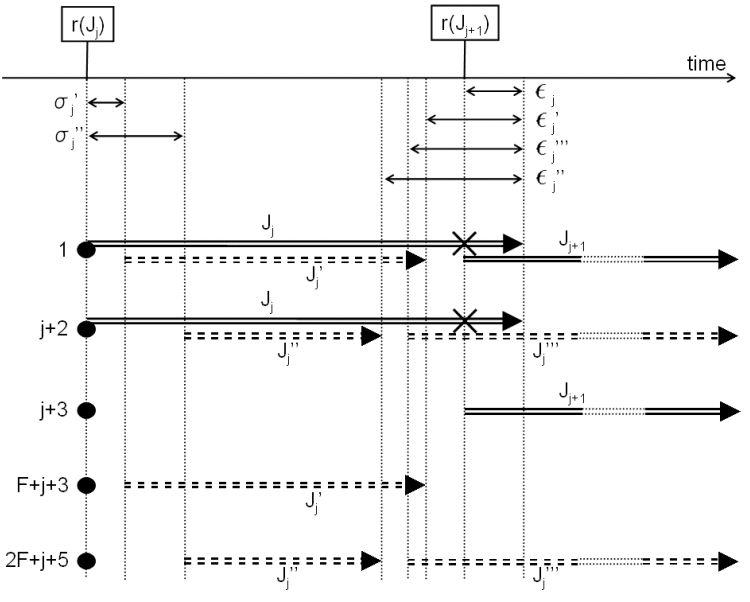


Fig. 2. A part of input sequence

5 The Case Without Slack Time

In this section, we study the case where each job has no slack time, namely, it must be either started immediately or rejected at its release time. In online job scheduling, this case has been studied in [4].

5.1 Overview of the Analysis

Let us classify jobs accepted by OPT or $GSMM(k)$ into three categories: A job only accepted by OPT is called an *o-extra job*. A job only accepted by $GSMM(k)$ is called a *g-extra job*. A job accepted by both OPT and $GSMM(k)$ is called a *common job*. Let GE_t and CM_t be the set of g-extra jobs and common jobs started by $GSMM(k)$ at time t, respectively. Note that EX denotes the set of o-extra jobs and $A_t = GE_t \cup CM_t$. Then, it is easy to see that $V_{OPT}(\sigma) = \sum_{t\in\mathcal{T}} p(CM_t) + p(EX)$ and $V_{GSMM(k)}(\sigma) = \sum_{t\in\mathcal{T}}(p(CM_t) + p(GE_t))$ for any σ. In the next section, in order to improve an upper bound, we show $p(EX) \le (2k+2)\sum_{t\in\mathcal{T}} p(GE_t) + \frac{k+2}{k-1}\sum_{t\in\mathcal{T}}(p(CM_t)+p(GE_t))$. Therefore, $V_{OPT}(\sigma) = \sum_{t\in\mathcal{T}} p(CM_t) + p(EX) \le \sum_{t\in\mathcal{T}} p(CM_t) + (2k+2)\sum_{t\in\mathcal{T}} p(GE_t) + \frac{k+2}{k-1}\sum_{t\in\mathcal{T}}(p(CM_t) + p(GE_t)) \le (2k+2+\frac{k+2}{k-1})V_{GSMM(k)}(\sigma)$ holds. We have that $2k+2+\frac{k+2}{k-1} = 2\sqrt{6}+5 \approx 9.90$ when $k = 1+\sqrt{6}/2$. Hence, we have the following theorem:

Theorem 5.1. *The competitive ratio of* $GSMM(1+\sqrt{6}/2)$ *is at most* $2\sqrt{6}+5$.

If we change the slack time of J_F into 0 in the lower bound example in Sec. 4, we obtain a lower bound of $GSMM(k)$ in this case.

Theorem 5.2. *For any positive constant* ε*, the competitive ratio of* $GSMM(k)$ *is at least* $2\sqrt{6}+5-\varepsilon$.

5.2 Evaluating $p(EX)$ with $p(GE)$ and $p(CM)$

The following lemma is immediate from Corollary 3.8.

Lemma 5.3

$$C(GE_t) \le (2k+2)p(GE_t).$$

Proof. Using Corollary 3.8, $C(A_t) \le (2k+2)p(A_t)$ holds. Since $GE_t \subseteq A_t$, $C(GE_t) \le (2k+2)p(GE_t)$.

Next, we estimate $C(CM_t)$.

Lemma 5.4

$$C(CM_t) = 0.$$

Proof. By the definition of CM_t, for any job $J \in CM_t$, OPT schedules J at time t when J is scheduled by $GSMM(k)$. As a result, J does not conflict with any job in EX.

Therefore, we have the following lemma.

Lemma 5.5

$$p(EX) \le (2k+2)\sum_{t\in\mathcal{T}} p(GE_t) + \frac{k+2}{k-1}\sum_{t\in\mathcal{T}}(p(CM_t)+p(GE_t)).$$

Proof. Using Lemma 3.3, Lemma 5.3, Lemma 5.4 and Corollary 3.9, $p(EX) \le (2k+2)\sum_{t\in\mathcal{T}} p(GE_t) + (k+2)\sum_{t\in\mathcal{T}} p(P_t)$ holds. By Lemma 3.11, we have shown that the statement is true.

Acknowledgement

We thank Associate Professor Shuichi Miyazaki for a lot of advice in this research.

References

1. Baruah, S., Koren, G., Mao, D., Mishra, B., Raghunathan, A., Rosier, L., Shasha, D., Wang, F.: On the competitiveness of on-line real-time task scheduling. Journal of Real-Time Systems 4(2), 125–144 (1992)
2. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press, Cambridge (1998)
3. Dolev, S., Kesselman, A.: Non-Preemptive Real-Time Scheduling of Multimedia Tasks. Journal of Real-Time Systems 17(1), 23–39 (1999)
4. Goldman, S., Parwatikar, J., Suri, S.: Online scheduling with hard deadlines. Journal of Algorithms 34(2), 370–389 (2000)
5. Goldwasser, M.: Patience is a virtue: the effect of slack on competitiveness for admission control. Journal of Scheduling 6(3), 183–211 (2003)
6. Guez, D., Kesselman, A., Rosén, A.: Packet-mode policies for input-queued switches. In: Proc. SPAA 2004, pp. 93–102 (2004)
7. Irani, S., Leung, V.: Scheduling with conflicts and applications to traffic signal control. In: Proc. SODA 1996, pp. 85–94 (1996)
8. Koren, G., Shasha, D.: D^{over}: an optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems. SIAM J. Comput. 24(2), 318–339 (1995)
9. Lee, J.H., Chwa, K.Y.: Online scheduling of parallel communications with individual deadlines. In: Aggarwal, A.K., Pandu Rangan, C. (eds.) ISAAC 1999. LNCS, vol. 1741, pp. 383–392. Springer, Heidelberg (1999)
10. Lipton, R.J., Tomkins, A.: Online interval scheduling. In: Proc. SODA 1994, pp. 302–311 (1994)
11. Sleator, D., Tarjan, R.: Amortized efficiency of list update and paging rules. CACM 28, 202–208 (1985)
12. Woeginger, G.J.: On-line scheduling of jobs with fixed start and end times. Theoretical Computer Science 130(1), 5–16 (1994)

Bundle Pricing with Comparable Items

Alexander Grigoriev[1], Joyce van Loon[1,⋆], Maxim Sviridenko[2], Marc Uetz[1], and Tjark Vredeveld[1]

[1] Maastricht University, Quantitative Economics,
P.O. Box 616, NL–6200 MD Maastricht, The Netherlands
{a.grigoriev,j.vanloon,m.uetz,t.vredeveld}@ke.unimaas.nl
[2] IBM T.J. Watson Research Center,
P.O. Box 218, Yorktown Heigths, NY 10598, USA
sviri@us.ibm.com

Abstract. We consider a revenue maximization problem where we are selling a set of items, each available in a certain quantity, to a set of bidders. Each bidder is interested in one or several bundles of items. We assume the bidders' valuations for each of these bundles to be known. Whenever bundle prices are determined by the sum of single item prices, this algorithmic problem was recently shown to be inapproximable to within a semi-logarithmic factor. We consider two scenarios for determining bundle prices that allow to break this inapproximability barrier. Both scenarios are motivated by problems where items are different, yet comparable. First, we consider classical single item prices with an additional monotonicity constraint, enforcing that larger bundles are at least as expensive as smaller ones. We show that the problem remains strongly NP-hard, and we derive a PTAS. Second, motivated by real-life cases, we introduce the notion of affine price functions, and derive fixed-parameter polynomial time algorithms.

1 Introduction

Consider the situation that we want to sell a set of items to a set of bidders. Every bidder places bids on subsets, or *bundles* of items, and each bidder would like to receive one or more of these bundles (OR-bids). Bidders have valuations for each of their bundles. The valuation is the maximum amount a bidder is willing to pay for a particular bundle. We assume that the bundles and the valuations are known. Hence we are faced with a purely algorithmic problem, in contrast to a mechanism design problem where the valuations are private information to the bidders. We have a certain amount of copies of each item available, and this amount may be limited or unlimited, for example non-digital or digital goods. We need to determine two things, namely which of the bidders receive which of their requested bundles, and how much each of them needs to pay. The goal is to maximize the total revenue received from the bidders. A

⋆ Supported by METEOR, the Maastricht Research School of Economics of Technology and Organizations.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 475–486, 2007.

general economic constraint on the possible prices, adopted in this paper as well, is that of *envy-freeness*. It requires that no bidder is left envious in the sense that she could afford a bundle, but doesn't receive it[1]. This is the general setting for the pricing problems studied in this paper.

In a sequence of recent papers [1,2,4,5,8,9,10], several algorithms and complexity results have been derived for such price optimization problems. The pricing model that is assumed in all these papers is the problem with *single item prices*, where each item is assigned an anonymous price, and bundle prices are defined by the sum of the respective item prices. We contribute to this line of research in two different directions.

First, we consider the single item prices model. We introduce a monotonicity constraint that allows us to derive results that break the semi-logarithmic inapproximability barrier known for the general case [5]. We impose the condition that the price of any bundle of size k must not exceed the price of a set of size $k+1$ or larger, for any k. This condition implies that (most of) the items for sale are *comparable* in the sense that the prices do not differ too much.

Second, we propose a model for determining bundle prices that actually generalizes the single item prices problem. We derive fixed-parameter polynomial time algorithms for that model. We assume that the bundle prices are determined on the basis of arbitrary affine functions, defined on a joint set of variables. Let us give an illustrating example: For each bundle j, there is an individual fixed cost a_j that the bidder needs to pay when purchasing that bundle. All items are identical and the seller just needs to determine one per-item price, say x. The price for any bundle j of size k_j is then given by $a_j + k_j x$. Notice that the price paid for any bundle is an affine function that depends on the size of the requested bundle. In general, the affine pricing model allows for many more pricing scenarios relevant in practice, e.g. groups of customers with different price functions, quantity discounts, etc. See Section 3 for a brief discussion.

Before we elaborate on related work and our contribution, let us define the pricing settings more formally.

1.1 Model

Let $I = \{1, \ldots, m\}$ denote the set of comparable items for sale, and let $J = \{1, \ldots, n\}$ denote the set of bids placed by all bidders. Each bid $j \in J$ is on exactly one subset of items $I_j \subseteq I$. In line with notation in auction literature, we call the set I_j also a *bundle*. Every bidder has a positive valuation for each of her bundles, that is, every bundle I_j corresponding to bid $j \in J$, has a positive valuation b_j which is the maximum amount its bidder is willing to pay for bundle I_j. We may assume w.l.o.g. that $b_j \geq 1$ for $j \in J$. The valuations are assumed to be known to the seller. Let c_i denote the available number of copies of item $i \in I$. We consider both the case of unlimited availability of items, that is, $c_i \geq n$ for all items $i \in I$, and the case of limited availability of items.

[1] More generally, envy-freeness requires that in an allocation, the bundle allocated to a bidder belongs to her *demand set*, which is the set of all allocations that maximize the bidder's utility [11].

A bid is a *winning bid* if it is assigned to the bidder, and a *losing bid* otherwise. The set of winning bids is denoted by $W \subseteq J$. A solution to the problem is a price $p(j)$ for the bundle I_j corresponding to bid $j \in J$. Later we will be more specific about further restrictions on the prices. A solution is called *feasible* if the bundles of all winning bids can be afforded by the respective bidder (that is, the price of the bundle corresponding to a bid is at most its valuation), and if no item is oversold. A solution is *envy-free* if in addition, for all losing bids the respective bundle is priced higher than the valuation of the corresponding bid. Let us summarize the above discussion in a definition for the generic pricing problem that we address in the paper.

Definition 1. *A feasible and envy-free solution to a pricing problem consists of a price $p(j)$ for the bundle I_j, for all bids $j \in J$, and a set of winning bids $W \subseteq J$ that are assigned to their bidders such that*

1. *the bundle of every winning bid j is affordable for the bidder, that is $p(j) \leq b_j$ for all $j \in W$,*
2. *the bundle of every losing bid j is too expensive for the bidder, that is $p(j) > b_j$ for all $j \in J \setminus W$,*
3. *no item is oversold, that is $\sum_{j \in W} |\{i\} \cap I_j| \leq c_i$ for all items $i \in I$.*

The objective is to find a solution that maximizes the total revenue of the seller, that is, we want to maximize $\sum_{j \in W} p(j)$.

We consider two different models for the computation of prices. The first pricing model that we consider is the single item prices model, where we have to determine item prices for all items $i \in I$. To be in line with previous papers on the same topic, let p_i denote the price of item i, and the price of bundle $I_j \subseteq I$ is $p(j) = \sum_{i \in I_j} p_i$, for all $j \in J$. Given that several inapproximability results exist for this model [5,8,9], we introduce a monotonicity constraint on the set of item prices. Specifically, we impose that the following holds true for any two subsets of items I' and I''.

$$p(I') \leq p(I'') \text{ whenever } |I'| < |I''| \,. \tag{1}$$

The condition has a meaningful economic interpretation in a lot of settings where items are different yet comparable, as it only requires that larger bundles are at least as expensive as smaller ones. It yields that (most) item prices are of the same order of magnitude. We show how this monotonicity constraint can be exploited to derive results that break the inapproximability barrier known for the general unconstrained case.

In practice, bundle prices are often not determined by the sum of individual item prices, but rather by a function based on a few bundle characteristics. Therefore, we propose the second model, in fact generalizing the single item prices problem, see Proposition 1. Here, the price of bundle I_j is determined by an affine function in some dimension K as follows.

$$p(j) = a_{j0} + a_{j1}x_1 + \cdots + a_{jK}x_K \,, \qquad j \in J \,. \tag{2}$$

The coefficients a_{jk}, $k = 0, \ldots, K$, are arbitrary coefficients that are given for all bids $j \in J$. These coefficients may, in general, depend on both the bundle I_j and on the bidder that places bid j. Thus it may be the case that similar bundles have different prices. The pricing problem consists of determining revenue-maximizing values for the (nonnegative) variables x_k, $k = 1, \ldots, K$. We postpone the discussion of this model to Section 3.

In the remainder of this paper, we denote by a ρ-approximation algorithm an algorithm that produces a solution with value at least $1/\rho$ times the optimal solution value. A PTAS is a family of $(1+\varepsilon)$-approximation algorithms, for any $\varepsilon > 0$.

1.2 Related Work

The problem mainly addressed in the literature is the one with unlimited availability of items, single item prices, and the requirement that the solution is envy-free [1,2,4,5,9,10]. For this problem the maximum revenue is hard to approximate to within a semi-logarithmic factor in the number of bids n [5]. In particular, it is unlikely that a constant approximation algorithm exists. For the same problem, Hartline and Koltun [10] present an approximation scheme with almost linear running time, given that the number of distinct items is constant. Moreover, given that each bidder is interested in bundles of at most k items, Balcan and Blum [2] derive an $O(k)$-approximation. Finally, there exist two fully polynomial time approximation schemes [2,4] for the problem where the bundles are *nested*, that is, for any two bundles I_j and $I_{j'}$ it holds that $I_j \subseteq I_{j'}$, $I_{j'} \subseteq I_j$ or $I_j \cap I_{j'} = \emptyset$.

1.3 Our Results

For the revenue maximization problem with single item prices, we derive strong NP-hardness even if prices need to fulfill the monotonicity constraint. Moreover, we derive a PTAS for that problem, with a time complexity of $O(nm^{8/\varepsilon}(\log B))^{8/\varepsilon})$, where $B = \max_j b_j$.

For the revenue maximization problem with affine price functions, we propose an algorithm with a time complexity of $O((K^3+nK)(n+K)^K)$. Here, parameter K is the dimension in which the affine price functions live. In particular, for $K = 1$ this is $O(n^2)$. A similar result (with slightly different time complexity) holds for the case of limited availability of items. For the same problem with nonconstant K, and unlimited availability of items, the maximum revenue is hard to approximate to within a semi-logarithmic factor in the number of bids n. This follows directly from the corresponding result by Demaine et al. [5], as we can show that the problem with single item prices is a special case. In addition, for the same pricing problem with limited availability of items, we prove that it is even NP-complete to approximate the maximum revenue to within a factor of $n^{1-\varepsilon}$ of optimum, where n is the number of bids.

A special case of single item pricing is the so-called highway pricing problem as suggested in [9]. There the bundles are subpaths of a simple path. We show that

this problem remains NP-hard even under the monotonicity assumption, and we derive a simple $O(\log B)$-approximation algorithm, where $B = \max_{j \in J} b_j$.

2 Single Item Pricing with Monotonicity Constraint

In single item pricing, we need to assign an (anonymous) item price p_i for each item $i \in I$, and bundle prices $p(j)$ are defined as the sum of the prices of the requested items, $p(j) = \sum_{i \in I_j} p_i$, for all $j \in J$. As before, the item prices need to yield a feasible and envy-free solution, and we wish to maximize the total revenue, which can be written as $\sum_{j \in W} \sum_{i \in I_j} p_i$. Notice that in case of unlimited availability of items both feasibility and envy-freeness is in fact no issue – yet finding optimal prices is hard [5]. For this reason we introduce a *monotonicity constraint*: $p(I') \leq p(I'')$ if $|I'| < |I''|$, for any two subsets of items I' and I''.

2.1 Complexity

Theorem 1. *The revenue maximization problem with single item prices and unlimited availability of items is strongly NP-hard, even if the prices need to satisfy the monotonicity constraint.*

Proof. We use a reduction from the strongly NP-hard problem IndependetSet [6]. Let $G = (V, E)$ be a graph in which we want to find a maximum cardinality set of vertices that are pairwise non-adjacent. Let M be an integer that is large enough. For every vertex $v \in V$ we create a *vertex-item*, and for every edge $e \in E$ we introduce an *edge-item*, that is, $I = V \cup E$. For every item $i \in I$, there are $M + 2$ bids placed on the bundle consisting of only this item. One of these bids has valuation M, and the others have the same valuation $M + 1$. Moreover, for every edge $e = \{u, v\} \in E$, there are four more bids. One bid is on bundle $\{u, e\}$, one bid on bundle $\{v, e\}$, and two bids are on bundle $\{u, v\}$. These four bids each have valuation $2M + 1$.

We claim that there exists an independent set of size s in G if and only if there is a solution for the revenue maximization problem with revenue $f + s$, where f is a function of M, $|V|$ and $|E|$. To prove this claim, we show that in any optimal solution all items are priced either at M or at $M + 1$. Moreover, vertex-items are priced at $M + 1$ if and only if the corresponding vertex belongs to the independent set. Details of the proof are included in the full version of this paper. □

2.2 Approximation Scheme

In order to derive a PTAS for the problem with single item prices and monotonicity constraint, we restrict the prices to powers of $(1 + \delta)$ for some $\delta > 0$. Assume, without loss of generality, that $p_1 \leq p_2 \leq \ldots \leq p_m$, then by the monotonicity constraint, we know $2p_2 \geq p_1 + p_2 \geq p_m$. Similarly, $3p_3 \geq p_1 + p_2 + p_3 \geq p_{m-1} + p_m \geq 2p_{m-1}$, etc.

Lemma 1. *Suppose $p_1 \leq p_2 \leq \ldots \leq p_m$. Any pricing of the items satisfying the monotonicity constraint also satisfies*

$$k\, p_k \geq (k-1) p_{m-k+2}, \quad k = 2, \ldots, \left\lceil \frac{m}{2} \right\rceil . \tag{3}$$

The idea for the PTAS is now the following. Except for a constant number of the cheapest and most expensive items, all items have prices in roughly the same range. Therefore we can price all except a constant number of items uniformly with the same price, without loosing too much in terms of the total revenue. We therefore enumerate over all possible uniform prices for the bulk of the items, and over all possible combinations of prices for the remaining (constant number of) items.

Theorem 2. *The pricing problem with unlimited availability of items, single item prices and monotonicity constraint admits a PTAS. The time complexity is $O(nm^{8/\varepsilon}(\log B)^{8/\varepsilon})$, where ε is the precision of the PTAS and $B = \max_j b_j$.*

Proof. Given an instance of the pricing problem and an $\varepsilon > 0$, let $\delta = \varepsilon/4$, and for convenience assume that $1/\delta$ is integral. Assume that we know the order of prices, say $p_1 \leq \cdots \leq p_m$, in an optimum solution. Define the subsets of items $S = \{i \in I : i \leq \frac{1}{\delta}\}$, $M = \{i \in I : 1 + \frac{1}{\delta} \leq i \leq m + 1 - \frac{1}{\delta}\}$ and $L = \{i \in I : i \geq m + 2 - \frac{1}{\delta}\}$. Note that $M = \emptyset$ if $\varepsilon \leq 8/(m+1)$, in which case the number of items is in $O(1/\varepsilon)$. We round down the prices of all items in S and L to powers of $(1+\delta)$. Moreover, we price all items in M uniformly at price $p_{1+1/\delta}$, rounded down to a power of $(1+\delta)$. Let us call the new prices p', and let us call p'_M the price of items in M. First observe that the order of prices does not change. We next argue that we do not loose too much by this rounding. Clearly, since we only round down, the set of winning bids can only increase. Moreover, we loose at most a factor $(1+\delta)$ on items in S and L. Finally, consider the items in M. By (3), we have

$$\left(1 + \frac{1}{\delta}\right) p_{1+1/\delta} \geq \frac{1}{\delta} p_{m+1-1/\delta} .$$

In other words, the price for the most expensive item in M differs from the cheapest item in M by a factor at most $(1+\delta)$. Hence, on items in M we loose a factor at most $(1+\delta)^2$.

Now we have a structured solution, but it may violate the monotonicity constraint. We claim that any such violation can be restored by one more rounding operation, if necessary: We just round down the price of all items priced p'_M or higher by another factor $(1+\delta)$. For contradiction, after this last rounding consider two violating sets I' and I'' with $|I'| < |I''|$ and $p'(I') > p'(I'')$, and w.l.o.g. $|I'| = \ell$, and $|I''| = \ell + 1$. Due to the ordering of prices, we then also have that $p'(\{1, \ldots, \ell+1\}) < p'(\{m, m-1, \ldots, m-\ell+1\})$. As long as there are items from M in both sets, we redefine $\ell = \ell - 1$, and we keep violating the monotonicity constraint. But now, all items in $\{1, \ldots, \ell\}$ have been rounded down by a factor at most $(1+\delta)$, and all items in $\{m, m-1, \ldots, m-\ell+1\}$ have

been rounded down by a factor at least $(1+\delta)$. This contradicts the monotonicity constraint of the optimal solution that we started with.

The PTAS now consists of enumerating all possible structured solutions, which is sufficient to obtain a feasible solution that differs from the optimal solution by a factor at most $(1+\delta)^3 < (1+\varepsilon)$. There are $\binom{m}{-1+2/\delta}$ possible choices for $S \cup L$. Since all prices are powers of $(1+\delta)$, there are $\log B$ possible prices. Given that all items in M have the same price, there are at most $(\log B)^{2/\delta}$ structured solutions for each choice of $S \cup L$. Computation of the revenue for any such solution takes $O(nm)$ time. This together with $\delta = \varepsilon/4$ yields the claimed time complexity, where the constant hidden in the O-notation depends on ε. □

3 Pricing with Affine Price Functions

For this section, we address revenue maximization problems with bundle prices $p(j)$ that are determined via affine price functions $p(j) = a_{j0} + a_{j1}x_1 + \cdots + a_{jK}x_K$, one affine function per bid $j \in J$, as defined in (2).

Let us discuss examples to motivate this model. If we let $K = 1$ and define $a_{j1} = |I_j|$ for all bids j, the bundle prices are determined by affine functions that depend only on the size of the bundles. The optimization problem is to determine the per-item price x_1, which is identical for all items. Fixed costs per bid j are incorporated by letting $a_{j0} \neq 0$. Bidder-dependent characteristics are easily incorporated as well, for example cost reductions of $\alpha_t\%$ for certain types t of customers, e.g. letting $a_{j2} = -\alpha_t a_{j1}$. Another meaningful interpretation is this: There are K different item types $i = 1, \ldots, K$ for which we need to determine the per-item prices x_i, and any bid j is specified by the number of requested items of type i, a_{ji}, and the total valuation b_j. In fact, one motivation for this model is phone contracts, where x_1 represents the monthly subscription fee ($a_{j1} = 1$ for all j), x_2 is the price per SMS and x_3 is the price per minute for phone calls. Here, the coefficients a_{j2} and a_{j3} describe typical average usages for different types j of customers.

We distinguish between unlimited and limited availability of items.

3.1 Unlimited Availability of Items

Our algorithm is polynomial as long as K, the dimension of the affine price functions is constant. It simply enumerates all vertices of the linear arrangement defined by the *valuation constraints*. Here, the valuation constraints are given by $p(j) \leq b_j$ for every winning bid j. More precisely, suppose that we know which of the bids are winning bids in an optimum solution, say $W \subseteq J$. Then we know that the variables $x_1, \ldots, x_K$ have to fulfill the $|W|$ inequalities

$$a_{j0} + a_{j1}x_1 + \cdots + a_{jK}x_K \leq b_j\,, \quad j \in W\,.$$

Denote by $\mathcal{P}$ the polyhedron defined by these $|W|$ inequalities. For an optimum solution $x = (x_1, \ldots, x_K)$, at least one of these inequalities must be tight, because otherwise the bundles corresponding to the same set of winning bids could be

priced even higher. Assume that $W' \subseteq W$ are the bids for which the above inequalities are tight, and note that W' is nonempty. Then the system

$$a_{j0} + a_{j1}x_1 + \cdots + a_{jK}x_K = b_j\,, \quad j \in W'$$

defines a (nonempty) face $\mathcal{F}$ of the polyhedron $\mathcal{P}$. By definition, any point $x \in \mathcal{F}$ defines an optimal solution. Clearly, at most K inequalities are required to completely characterize the face $\mathcal{F}$. Moreover, we have exactly $\dim(\mathcal{F})$ free variables in the optimal solution x. In other words, the same total revenue can be obtained by fixing the $\dim(\mathcal{F})$ free variables among $x_1, \ldots, x_K$ to 0. Hence, an optimal solution can be obtained by considering all solutions x that are characterized by K linearly independent constraints out of the following $n + K$ constraints.

$$a_{j0} + a_{j1}x_1 + \cdots + a_{jK}x_K = b_j\,, \qquad j \in J\,, \tag{4}$$

$$x_k = 0\,, \qquad k = 1, \ldots, K\,. \tag{5}$$

This insight can be used to define a simple algorithm that solves the revenue maximization problem in polynomial time, as long as K is constant.

Algorithm 1. Revenue maximization with affine price functions

Input: Instance with affine price functions as defined in (2).

For all candidate solutions $x = (x_1, \ldots, x_K)$ that fulfill K linearly independent constraints out of the $n + K$ constraints (4) and (5) do:
- –Let $p(j) = a_{j0} + a_{j1}x_1 + \cdots + a_{jK}x_K$, $j \in J$, be the bundle prices.
- –Let $W := \{j \in J : p(j) \leq b_j\}$ be the set of winning bids.
- –Let $\Pi = \sum_{j \in W} p(j)$ be the total revenue.

Output: Maximum among all values Π, with optimal parameters $x_1, \ldots, x_K$, and set of winning bids W.

Theorem 3. *Algorithm 1 solves the revenue maximization problem with affine price functions and unlimited availability of items in* $O((K^3 + nK)(n + K)^K)$ *time.*

Proof. Correctness of the algorithm immediately follows from the preceding discussion. We need to consider $\binom{n+K}{K} \in \mathrm{O}((n + K)^K)$ systems of K constraints each. In each of these iterations, we need to solve a linear system in K variables and K constraints, which takes $\mathrm{O}(K^3)$ time. Computation of the bundle prices, winning bids, and the objective value takes $\mathrm{O}(nK)$ time. The claimed time complexity follows. □

In contrast, if the dimension K of the price functions is not constant, the problem is much harder. In fact, if K is not constant, the single item prices problem is just a special case of the problem with affine price functions: Let $K = m$, and let $a_{ji} = 1$ whenever item i is contained in bundle I_j and $a_{ji} = 0$ otherwise. Then, each item price corresponds to one variable x_i, and we immediately obtain the following.

Proposition 1. *The model with affine price functions generalizes the model with single item prices. Hence, the semi-logarithmic inapproximability result by Demaine et al. [5] also holds for pricing problems with affine price functions.*

3.2 Limited Availability of Items

First we claim that Algorithm 1 can as well be used to solve the problem when the availability of items is limited. Indeed, the only thing we additionally need to check for any of the candidate solutions is feasibility: We have to verify whether none of the items is oversold, and if yes, we do not consider the candidate solution. Also note that any candidate solution is envy-free by definition. Clearly, this feasibility check can be done in $O(nm)$ time per candidate solution.

Corollary 1. *Algorithm 1, augmented with a feasibility check, solves the revenue maximization problem with affine price functions and limited availability of items in $O((K^3 + nK + nm)(n + K)^K)$ time.*

On the negative side, it turns out that the problem with limited availability of the items seems even harder to approximate.

Theorem 4. *Consider the revenue maximization problem with affine price functions and limited availability of items. For any $\varepsilon > 0$, it is NP-hard to approximate the maximum revenue to within a factor $n^{1-\varepsilon}$. This result holds even if all bids have a valuation of one, the availability of each item is one, and each item is requested in at most two bids.*

Proof. We use an approximation preserving reduction from IndependetSet. Given is a graph $G = (V, E)$, we want to find a maximum cardinality subset $V' \subseteq V$ such that no two vertices in V' are adjacent. Zuckerman [12] showed that it is NP-hard to approximate the maximum independent set to within a factor $|V|^{1-\varepsilon}$, for any $\varepsilon > 0$.

We construct the following instance of the pricing problem. Each vertex $v \in V$ corresponds to a bid and each edge $e \in E$ corresponds to an item. Each bid v is on a bundle containing all edges incident to v, and has valuation $b_v = 1$. Each item e is available once ($c_e = 1$). Let the price functions be $p(v) = 1 + x_v$, for all bundles I_v corresponding to bid $v \in V$.

We claim that an independent set of cardinality s exists in G if and only if there exists a pricing for the above defined instance with total revenue s. Suppose $V' \subseteq V$ is an independent set in G, $|V'| = s$. Then let $x_v = 0$ for all $v \in V'$, and $x_v > 0$ otherwise. This way the set of winning bids equals the independent set V', and therefore no item is oversold. No bidder is envious, as the price of a bundle corresponding to a losing bid exceeds its valuation, and we extract a total revenue of s.

Conversely, assume a solution to the pricing problem with total revenue s. Since only one copy of any item is available, the set of winning bids must define an independent set in G. As the maximum revenue from any bidder's bid is 1, there exists an independent set of size s in G. □

4 Highway Problem with Monotonicity Constraint

A particularly intriguing special case of the single item prices problem is the 'highway problem' as introduced by Guruswami et al. [9]. Here, the items are edges of a simple path, and the bundles corresponding to bids requested by bidders are subpaths. The problem is NP-hard [4] by a simple transformation from PARTITION, and a $\log(m)$-approximation exists [2]. In this setting, it is natural to assume that the monotonicity constraint holds for any two *subpaths* only, but not necessarily for arbitrary subsets of items. For this problem we obtain the following results.

Theorem 5. *The highway problem with monotonicity constraint is NP-hard.*

The proof of this theorem is deferred to the full version of this paper. Notice that we cannot apply the PTAS from Theorem 2, as this crucially requires the monotonicity constraint for arbitrary subsets of items. Nevertheless, we derive an $O(\log B)$-approximation algorithm for the highway pricing problem with monotonicity constraint, where $B = \max_j b_j$. To this end, we present approximation guarantees for two special cases first.

Lemma 2. *The highway pricing problem with monotonicity constraint in which all bundles have size at least two is approximable within a factor of* 3 *by optimal uniform pricing.*

Proof. Consider an optimal solution with revenue OPT and let $p^*_{\max}$ be the highest item price in this solution. We claim that pricing all items at $p^*_{\max}/3$, yields a revenue of at least OPT/3. Clearly, an optimal uniform pricing is at least as good as the uniform $p^*_{\max}/3$ pricing.

First, we show that any winning bid $j \in W$ in the optimal pricing remains a winning bid for the uniform pricing at level $p^*_{\max}/3$. Let $|I_j| = \ell$. Then the valuation for bid j is at least $b_j \geq \lfloor \ell/2 \rfloor p^*_{\max}$, as by the monotonicity constraint the total price of any two consecutive items in an optimal solution is at least $p^*_{\max}$ and the bidder who placed bid j can afford the corresponding bundle I_j. In the uniform $p^*_{\max}/3$ pricing, the total bundle price is $\ell p^*_{\max}/3$, which is at most $\lfloor \ell/2 \rfloor p^*_{\max}$, for $\ell \geq 2$. In an optimal pricing, bundle I_j corresponding to bid j is priced at most at $\ell p^*_{\max}$, whereas in our uniform pricing, we get $\ell p^*_{\max}/3$. Hence, pricing all items $p^*_{\max}/3$ yields a revenue of at least OPT/3. □

The above lemma shows that whenever all bundles contain at least two items, we have a constant approximation. Now, we consider only instances in which bundles consist of exactly one item. Moreover, we restrict ourselves to instances in which $b_j/b_k \leq 2$, for any two bids j and k.

Property 1. Consider the highway pricing problem with monotonicity constraint, restricted to instances in which each bid is on a bundle with exactly one item and for any two bids j, k, it holds true that $b_j/b_k \leq 2$. Pricing each item at $\min_j b_j$ yields a revenue of at least OPT/2.

Theorem 6. *The best uniform pricing yields a solution with revenue at least* $\textsc{opt}/(3+2\log_2 B)$ *for the highway pricing problem with monotonicity constraint, where* $B = \max_j b_j$*. Moreover, the time needed to find this solution is* $O(n^2 m)$*.*

Proof. Consider an optimum solution satisfying the monotonicity constraint, and let $\textsc{opt}_L$ denote the revenue of bidders whose bids are on bundles of size at least two and let $\textsc{opt}_r$ denote the revenue of bidders whose bids are on bundles of size one with valuation $2^{r-1} \leq b_j < 2^r$ $(r = 1, \ldots, \lceil \log_2 B \rceil + 1)$ in this solution. Then $\textsc{opt} = \textsc{opt}_L + \sum_r \textsc{opt}_r$.

Moreover, let $\textsc{app}_L$ denote the revenue obtained by the best uniform pricing and $\textsc{app}_r$ denote the revenue obtained by the best uniform pricing strategy for the bidders with bids in $J_r = \{j \in J : |I_j| = 1 \text{ and } 2^{r-1} \leq b_j < 2^r\}$. By Property 1, we have that $\textsc{app}_r \geq \textsc{opt}_r/2$ and thus $\max_r \textsc{app}_r \geq \sum_r \textsc{opt}_r/(2\log_2 B)$. Moreover, from Lemma 2, it follows that $\textsc{app}_L \geq \textsc{opt}_L/3$. Hence, the solution found yields a revenue of

$$\max\{\textsc{app}_L, \textsc{app}_r : r = 1, \ldots, \lceil \log_2 B \rceil + 1\} \geq \textsc{opt}/(3 + 2\log_2 B).$$

To see the claim on the time complexity, note that to find sufficiently good uniform pricing, we need to consider at most $O(n)$ different prices. For each price, we need to compute the set of winning bids and the revenue obtained on this price, which can be done in $O(nm)$ time. So, the best uniform price can be computed in $O(n^2 m)$ time. □

5 Conclusion

This paper studies purely algorithmic, or *omniscient* pricing problems, reflected by the fact that we assume bidders' valuations b_j to be known. Even more challenging are problems where valuations are private information, and incentive-compatible mechanisms are sought, that is, mechanisms that induce bidders to truthfully report their valuations. To that end, one might ask if previous ideas on the design of (random sampling) mechanisms, e.g. by Goldberg et al. [7] or Balcan et al. [3], can be applied. Particularly the latter paper suggests a general approach for reducing incentive-compatible mechanism design problems to the underlying algorithmic pricing problems. Among them, combinatorial auctions with single-minded bidders (the model considered in this paper). We leave this on the agenda for future research; complications might lay in the monotonicity constraint that we impose on the pricing function, respectively the general form of the affine price functions.

Acknowledgements

We thank the anonymous referees for some inspiring and helpful remarks.

References

1. Aggarwal, G., Feder, T., Motwani, R., Zhu, A.: Algorithms for multi-product pricing. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 72–83. Springer, Heidelberg (2004)
2. Balcan, M.F., Blum, A.: Approximation algorithms and online mechanisms for item pricing. In: Proc. 7th ACM Conf. Electronic Commerce, pp. 29–35. ACM, New York (2006)
3. Balcan, M.F., Blum, A., Hartline, J.D., Mansour, Y.: Mechanism design via machine learning. In: Proc. 46th IEEE Found. Comp. Sci., FOCS 2005, pp. 605–614 (2005)
4. Briest, P., Krysta, P.: Single-minded unlimited supply pricing on sparse instances. In: Proc. 17th ACM-SIAM Symp. Discr. Alg., SODA 2006, pp. 1093–1102 (2006)
5. Demaine, E.D., Feige, U., Hajiaghayi, M.T., Salavatipour, M.R.: Combination can be hard: Approximability of the unique coverage problem. In: Proc. 17th ACM-SIAM Symp. Discr. Alg., SODA 2006, pp. 162–171 (2006)
6. Garey, M.R., Johnson, D.S.: Computers and intractability: A guide to the theory of np-completeness. W. H. Freeman, New York (1979)
7. Goldberg, A.V., Hartline, J.D., Karlin, A.R., Saks, M., Wright, A.: Competitive auctions. Games and Economic Behavior 55, 242–269 (2006)
8. Grigoriev, A., van Loon, J., Sitters, R., Uetz, M.: How to sell a graph: Guidelines for graph retailers. In: Fomin, F.V. (ed.) WG 2006. LNCS, vol. 4271, pp. 125–136. Springer, Heidelberg (2006)
9. Guruswami, V., Hartline, J.D., Karlin, A.R., Kempe, D., Kenyon, C., McSherry, F.: On profit-maximizing envy-free pricing. In: Proc. 16th ACM-SIAM Symp. Discr. Alg., SODA 2005, pp. 1164–1173 (2005)
10. Hartline, J.D., Koltun, V.: Near-optimal pricing in near-linear time. In: Dehne, F., López-Ortiz, A., Sack, J.-R. (eds.) WADS 2005. LNCS, vol. 3608, pp. 422–431. Springer, Heidelberg (2005)
11. Walras, L.: Elements of pure economics. Allen and Unwin, London (1954)
12. Zuckerman, D.: Linear degree extractors and the inapproximability of max clique and chromatic number, ECCC Report TR05-100 (2005), http://www.eccc.uni-trier.de/eccc/

Approximating Interval Scheduling Problems with Bounded Profits

Israel Beniaminy[1], Zeev Nutov[2], and Meir Ovadia

[1] ClickSoftware Technologies
israel@clicksoftware.com
[2] The Open University of Israel, Raanana, Israel
nutov@openu.ac.il, meiro@cadance.com

Abstract. We consider the *Generalized Scheduling Within Intervals* (GSWI) problem: given a set J of jobs and a set $\mathcal{I}$ of intervals, where each job $j \in J$ has in interval $I \in \mathcal{I}$ length (processing time) $\ell_{j,I}$ and profit $p_{j,I}$, find the highest-profit feasible schedule. The best approximation ratio known for GSWI is $(1/2-\varepsilon)$. We give a $(1-1/e-\varepsilon)$-approximation scheme for GSWI with bounded profits, based on the work by Chuzhoy, Rabani, and Ostrovsky [4] for the $\{0,1\}$-profit case. We also consider the *Scheduling Within Intervals* (SWI) problem, which is a particular case of GSWI where for every $j \in J$ there is a unique interval $I = I_j \in \mathcal{I}$ with $p_{j,I} > 0$. We prove that SWI is (weakly) NP-hard even if the *stretch factor* (the maximum ratio of job's interval size to its processing time) is arbitrarily small, and give a polynomial-time algorithm for bounded profits and stretch factor < 2.

1 Introduction

We consider the following problem:

Generalized Scheduling Within Intervals (GSWI)**:**

Instance: A set J of jobs and a set $\mathcal{I}$ of intervals, where each job $j \in J$ has in interval $I \in \mathcal{I}$ length $\ell_{j,I}$ and profit $p_{j,I}$, and each interval $I \in \mathcal{I}$ is given by $[r_I, d_I)$.

Objective: Find a maximum profit feasible schedule.

More precisely, a schedule S consists of a subset $J' \subseteq J$ of jobs, and for every $j \in J'$: an assignment to an interval $I(j) \in \mathcal{I}$ and a start time s_j, so that $[s_j, s_j + \ell_{j,I(j)}) \subseteq I(j)$. A schedule is feasible if the intervals $[s_j, s_j + \ell_{j,I(j)})$, $j \in J'$, are pairwise disjoint. The profit of such schedule is $p(S) = \sum_{j \in J'} p_{j,I(j)}$. We sometimes use S to denote only the corresponding set J' of jobs, if this does not cause ambiguity. Throughout this paper we assume that all the problem parameters are integers. Let $n = |J|$, $m = |\mathcal{I}|$, and let $P = \max_{j,I} p_{j,I}$.

GSWI and related problems have been used to model many applications of scheduling problems, including service and delivery, project planning, communication scheduling, space-mission-planning and optimization of production lines, c.f., [2,1,3,6]. The following three particular cases of GSWI were studied extensively.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 487–497, 2007.

- *Max-Profit Generalized Assignment* (MAX-GAP):
  the intervals in $\mathcal{I}$ are pairwise disjoint.
- *Scheduling Within Intervals* (SWI):
  for every $j \in J$ exactly one interval $I \in \mathcal{I}$ has $p_{j,I} > 0$.
- *Job Interval Selection Problem* (JISP):
  for every pair $j \in J, I \in \mathcal{I}$, either $p_{j,I} = 0$ or $\ell_{j,I} = |I|$.

Each one of these particular cases is strongly NP-hard [7]. Bar-Noy et al. [2] gave an approximation scheme for SWI (in fact, also for GSWI) with ratio of $1/2 - \varepsilon$, see also [1], and a faster algorithm due to Berman and DasGupta [3]. MAX-GAP admits a $(1-1/e)$-approximation algorithm [6], see also [8]. Recently, Feige and Vondrak [5] showed that MAX-GAP admits a ratio better than $(1-1/e)$. However, MAX-GAP is much "easier" than GSWI and the algorithm in [6,8,5] do not extend even to very special instances (e.g., $\{0,1\}$ profits and unit lengths) of SWI/JISP. Chuzhoy, Rabani, and Ostrovsky [4] considered JISP with $\{0,1\}$-profits, and gave for this case a randomized approximation scheme with ratio $(1-1/e-\varepsilon)$. We extend this result to arbitrary *bounded* profits. Our algorithm for GSWI uses as a subroutine an arbitrary constant ratio approximation algorithm SC for GSWI; let $1/\alpha$ be its approximation ratio and $Q = Q(n,m)$ its running time (e.g., the algorithm of [3] with $\alpha = 3$ has running time $Q(n,m) = O(n^2 m)$). A packing-type linear program is of the form $\max\{p \cdot y : yA \leq b, y \geq 0\}$, where A is an $n' \times m'$ $\{0,1\}$-matrix, and b is an m' $\{0,1\}$ vector. We assume that the time required to solve such program is $L(n', m')$.

Theorem 1. GSWI *admits a* $(1 - 1/e - \varepsilon)$*-approximation algorithm for any* $\varepsilon > 0$ *and constant* $P = \max_{j,I} p_{j,I}$*. The running time of the algorithm is* $O(n \cdot Q(n,m) \cdot \ln(P/\varepsilon) + (nm)^{q+1} L((nm)^{q+1}, n))$*, where* $q = 6k^{k \ln k + 3}$ *for* $k = \lceil (P + 2\alpha + 1)/\varepsilon \rceil = O(P/\varepsilon)$*, and* $L(n', m')$ *is the time required to solve a packing type linear program with* n' *variables and* m' *constraints.*

For $P = 1$ we get the algorithm of [4]. The running time, although polynomial for any $\varepsilon > 0$, is not practical. We leave an open question whether the time complexity can be reduced to be polynomial in the input size and $1/\varepsilon$, or whether GSWI admits a $(1 - 1/e)$-approximation algorithm. Such a better algorithm is known for MAX-GAP, but is not known even for the $\{0,1\}$-profit JISP/SWI.

Recall that in SWI every job $j \in J$ can be scheduled in a unique interval $I_j \in \mathcal{I}$ (as j has profit 0 in the other intervals); let j have length ℓ_j and profit p_j in I_j. Note that GSWI includes the "multiple machine" version of SWI. We consider SWI with bounded profits and small *stretch factor*, which is $\max_{j \in J} |I_j|/\ell_j$. Formally, let θ-SWI be the restriction of SWI to instances with $\max_{j \in J} |I_j|/\ell_j < \theta$, where $\theta > 1$. Berman and DasGupta [3] gave a pseudo-polynomial algorithm for θ-SWI with running time $O(\theta T n \log\log T)$ and with approximation ratio $1/2 + 1/(2^{a+2} - 4 - 2a)$, where $T = \max_j d_j$ and a is the largest integer strictly smaller than θ; this was used to derive an approximation scheme with ratio $(1/2 - \varepsilon) + 1/(2^{a+2} - 4 - 2a)$ and running time $O(n^2/\varepsilon)$. For 2-SWI $a = 1$ and thus the approximation ratio is $1 - \varepsilon$ (in [2] and in [3] 2-SWI was mistakenly mentioned to be in P, but this is so only for bounded profits). We prove:

Theorem 2. *θ-SWI is NP-hard for any $\theta > 1$ even if $p_j = \ell_j$ for every job $j \in J$ and all the time windows have the same length. 2-SWI can be solved in $O(\min\{n\mathcal{P}, rn^r \log n\})$ time, where $\mathcal{P} = \sum_{j \in J} p_j$ and r is the number of distinct profits.*

Theorems 1 and 2 are proved in Sections 2 and 3, respectively.

2 Proof of Theorem 1

Our algorithm extends the algorithm of [4] for $\{0,1\}$-profit JISP/SWI, to GSWI with any profits bounded by a constant P. Here is a high-level description of the algorithm. Let $[0,T)$ be the *timeline*, namely, the smallest interval that contains all the intervals in $\mathcal{I}$. The algorithm has two phases. In Phase I, the algorithm computes a partition $\mathcal{B}$ of $[0,T)$ into blocks (intervals) and:

(i) A schedule S^I within a subset $\mathcal{B}^I \subseteq \mathcal{B}$ of blocks (no job intersects a boundary of a block);
(ii) Subsets $J^{II} \subseteq J \setminus S^I$ of jobs and $B^{II} \subseteq \mathcal{B} \setminus \mathcal{B}^I$ of blocks.

The computed partition $\mathcal{B}$ has the property that the restriction "no job intersects a boundary of a block in $\mathcal{B}$" decreases the optimum by a small amount (if ε is small). We also have an upper bound on the profit of jobs from J^{II} scheduled by any optimal solution within any block in $\mathcal{B}^{II}$. In Phase II, the algorithm schedules jobs from J^{II} within blocks of $\mathcal{B}^{II}$ using an LP-relaxation created by enumerating all feasible schedules in each such block. LP-rounding gives a feasible schedule with expected approximation ratio $(1 - 1/e)$, and the algorithm is derandomized using the method of conditional expectations. The main differences between our algorithm and that of [4] is that in Phase I our algorithm partitions the timeline according to the *profit* of jobs we can schedule in each block, and that at Phase II we use a more general linear program.

Formally, let $\mathrm{OPT}_{\mathcal{B}}(J)$ be any optimal schedule of jobs from J under the constraint that no job's schedule intersects the boundary of a block from $\mathcal{B}$, and let $\mathsf{opt}_{\mathcal{B}}(J) = p(\mathrm{OPT}_{\mathcal{B}}(J))$ be its profit; for brevity $\mathrm{OPT}(J) = \mathrm{OPT}_{\{[0,T]\}}(J)$ and $\mathsf{opt}(J) = p(\mathrm{OPT}(J))$. Given a set J' of jobs and a block B, let $\mathrm{SC}(J', B)$ be any $1/\alpha$-approximation algorithm for scheduling jobs from J' within the block B, and let us denote by $Q = Q(n,m)$ its running time. As was mentioned, we may substitute $\alpha = 3$ and $Q(n,m) = O(n^2 m)$. Given $\varepsilon > 0$ let $k = \lceil (P + 2\alpha + 1)/\varepsilon \rceil$. We prove:

Lemma 1. *GSWI admits an algorithm that for any $\varepsilon > 0$ computes in time $O(n \cdot Q(n,m) \cdot \ln(P/\varepsilon))$ a partition $\mathcal{B}$ of $[0,T)$ into at most $n\varepsilon$ blocks, a schedule S^I within a subset $\mathcal{B}^I \subseteq \mathcal{B}$ of blocks, and subsets $J^{II} \subseteq J \setminus S^I$ of jobs and $\mathcal{B}^{II} \subseteq \mathcal{B} \setminus \mathcal{B}^I$ of blocks such that:*

(i) $p(S^I) + \mathsf{opt}_{\mathcal{B}^{II}}(J^{II}) \geq (1 - \varepsilon)\mathsf{opt}(J)$.
(ii) *In each block $B \in \mathcal{B}^{II}$, any optimal solution schedules at most $2\alpha \cdot k^{k \ln k + 3}$ jobs from J^{II}.*

An instance of GSWI is $(\mathcal{B},q)$-*restricted*, where $\mathcal{B}$ is a set of pairwise disjoint blocks in $[0,T)$ and q is an integer, if there exists an optimal solution that schedules all its jobs within the blocks of $\mathcal{B}$ with at most q jobs per block.

Lemma 2. *$(\mathcal{B},q)$-restricted* GSWI *admits a $(1-1/e)$-approximation algorithm with running time $O(|\mathcal{B}|(nm)^q \cdot L(|\mathcal{B}|(nm)^q, n+|\mathcal{B}|))$.*

Lemmas 1, 2 easily imply Theorem 1. Execute the algorithm as in Lemma 1 to compute $\mathcal{B}^I, \mathcal{B}^{II}, S^I, J^{II}$ as in the lemma. Then apply Lemma 2 on the $(\mathcal{B}^{II},q)$-restricted GSWI instance with $q = 2\alpha \cdot k^{k \ln k+3}$ and the set of jobs J^{II} to compute a schedule S^{II}. The running time is as claimed. Clearly, $S^I \cup S^{II}$ is a feasible solution, since $S^{II} \subseteq J \setminus S^I$ and since no block in $\mathcal{B}^I$ intersects a block in $\mathcal{B}^{II}$. The approximation ratio is as claimed since:

$$\begin{aligned} p(S^I)+p(S^{II}) &\geq p(S^I)+(1-1/e)\mathsf{opt}_{\mathcal{B}^{II}}(J^{II}) \\ &\geq (1-1/e)\left(p(S^I)+\mathsf{opt}_{\mathcal{B}^{II}}(J^{II})\right) \\ &\geq (1-1/e)(1-\varepsilon)\mathsf{opt}(J) \geq (1-1/e-\varepsilon)\mathsf{opt}(J)\ . \end{aligned}$$

2.1 Proof of Lemma 1

In the following procedure PARTITIONTIMELINE, in iteration i, $\mathcal{B}_i$ is the set of blocks partitioning the timeline $[0,T)$ and S_i is the schedule (or the set of jobs scheduled) within the blocks of $\mathcal{B}_i$; only these jobs are available for scheduling in the next iteration. Eventually, the algorithm returns a partition $\mathcal{B}$ of $[0,T)$ into blocks, a schedule S^I within a subset $\mathcal{B}^I \subseteq \mathcal{B}$ of blocks, and subsets $J^{II} \subseteq J \setminus S^I$ of jobs and $\mathcal{B}^{II} \subseteq \mathcal{B} \setminus \mathcal{B}^I$ of blocks.

Procedure PARTITIONTIMELINE(J,ε)
Initialization: $i \leftarrow 1$; $\mathcal{B}_0 \leftarrow \{[0,T)\}$; $S_0 \leftarrow J$; $k \leftarrow \lceil (P+2\alpha+1)/\varepsilon \rceil$.
Loop
  $S_i \leftarrow \emptyset$, $\mathcal{B}_i \leftarrow \mathcal{B}_{i-1}$
  For every block $B \in \mathcal{B}_i$ in ascending time order *do*:
    If $p(\mathrm{SC}(S_{i-1} \setminus S_i, B)) \geq k^{i+2}$ then *do*:
      - $S_i \leftarrow S_i \cup \mathrm{SC}(S_{i-1} \setminus S_i, B)$;
      - In ascending time order, scan the jobs scheduled by SC in B and partition B into blocks, each with largest possible profit $\leq k^{i+2}$, and add this partition to $\mathcal{B}_i$.
  EndFor
  Termination Condition 1: *If* $i = \lceil k \ln k \rceil$ then *do*:
    $S^I \leftarrow \emptyset$; $B^I \leftarrow \emptyset$;
    $J^{II} \leftarrow J \setminus S_i$; $\mathcal{B}^{II} \leftarrow \mathcal{B}_i$; STOP.
  Termination Condition 2: *If* $p(S_i) \geq (1-1/k)p(S_{i-1})$ then *do*:
    $S^I \leftarrow S_i$;
    $\mathcal{B}^I \leftarrow$ the blocks in $\mathcal{B}_i$ containing jobs from S^I;
    $J^{II} \leftarrow J \setminus S_{i-1}$; $\mathcal{B}^{II} \leftarrow \mathcal{B}_{i-1} \setminus \mathcal{B}^I$; STOP.
  Else (Termination Conditions 1,2 do not apply) $i \leftarrow i+1$.
EndLoop

The following statement uses only the fact that the number of iterations in PartitionTimeLine is $\lceil k \ln k \rceil \leq k^2$, and it is independent of the algorithm SC used.

Lemma 3. *$|\mathcal{B}| \leq |\mathrm{OPT}(J)|/k$ holds for the partition $\mathcal{B}$ produced by* PartitionTimeLine. *Thus $\mathsf{opt}_{\mathcal{B}}(J) \geq (1 - P/k)\mathsf{opt}(J)$.*

Proof. The number of new blocks created in iteration i is at most $p(S_i)/k^{i+2} \leq p(S_i)/k^3$. Each new block eliminates at most one job from OPT. Since all jobs in S_i can be scheduled, and every job in $\mathrm{OPT}(J)$ has profit at least 1 we have $|\mathrm{OPT}(J)| \geq |S_i|$. The maximum number of iterations is $\lceil k \ln k \rceil$. Therefore, the number of jobs eliminated from the optimal solution by all iterations is at most

$$\sum_{i=1}^{\lceil k \ln k \rceil} \frac{|S_i|}{k^3} \leq \frac{\lceil k \ln k \rceil}{k^3} |\mathrm{OPT}(J)| \leq \frac{1}{k} |\mathrm{OPT}(J)| \ .$$

The second statement follows from the first since every job has profit at most P.

In the rest of this section we prove the following lemma, that implies Lemma 1:

Lemma 4. *In each block $B \in \mathcal{B}_i$ computed by any iteration i of* PartitionTimeLine, *$\mathrm{OPT}(J)$ schedules jobs with at most total profit αk^{i+2} from $S_{i-1} \setminus S_i$.*

Proof. Consider two cases. In one case, block B may have been present in $\mathcal{B}_{i-1}$ and unmodified by iteration i. This could happen only if $\mathrm{SC}(S_{i-1} \setminus S_i, B)$ could not schedule jobs with total profits more than k^{i+2} in B. In the second case, block B was created by subdividing a block from $\mathcal{B}_{i-1}$ into blocks containing jobs with profit at most k^{i+2} by SC. In both cases, all the jobs from $S_{i-1} \setminus S_i$ were available for scheduling when SC started processing block B and SC gives $1/\alpha$-approximation.

Lemma 5. *$p(S^I) + \mathsf{opt}_{\mathcal{B}^{II}}(J^{II}) \geq (1 - \varepsilon)\mathsf{opt}(J)$.*

Proof. Consider the two termination conditions of PartitionTimeLine.

Termination Condition 1: PartitionTimeLine terminated after $\lceil k \ln k \rceil$ iterations, and $J^{II} = J \setminus S_i = J \setminus S_{\lceil k \ln k \rceil}$. For all iterations $1 \leq i < \lceil k \ln k \rceil$, the Termination Condition 2 was not satisfied, and thus $p(S_i) < (1 - 1/k)p(S_{i-1})$. Therefore

$$p(S_{\lceil k \ln k \rceil}) \leq (1 - 1/k)^{\lceil k \ln k \rceil} p(S_1) \leq p(S_1)/k \leq \mathsf{opt}(J)/k \ .$$

In this case, $\mathcal{B}^{II} = \mathcal{B}$. By Lemma 3, $\mathsf{opt}_{\mathcal{B}^{II}}(J) \geq (1 - P/k)\mathsf{opt}(J)$. Thus

$$\mathsf{opt}_{\mathcal{B}^{II}}(J^{II}) \geq \mathsf{opt}_{\mathcal{B}^{II}}(J) - p(S_{\lceil k \ln k \rceil}) \geq (1 - P/k)\mathsf{opt}(J) - \mathsf{opt}(J)/k \geq (1 - \varepsilon)\mathsf{opt}(J) \ .$$

Termination Condition 2: PartitionTimeLine terminated at iteration $i < \lceil k \ln k \rceil$, because the condition $p(S_i) \geq (1 - 1/k)p(S_{i-1})$ was satisfied. In this

case, $S^I = S_i$, $J^{II} = J \setminus S_{i-1}$, $\mathcal{B}^I$ includes all the blocks containing jobs from S^I, and $\mathcal{B}^{II} = \mathcal{B}_{i-1} \setminus \mathcal{B}^I$. Thus:

$$\begin{aligned}\mathsf{opt}_{\mathcal{B}^{II}}(J^{II}) &\geq \mathsf{opt}_{\mathcal{B}_{i-1}}(J^{II}) - \mathsf{opt}_{\mathcal{B}^I}(J^{II}) = \\ &= \mathsf{opt}_{\mathcal{B}_{i-1}}(J \setminus S_{i-1}) - \mathsf{opt}_{\mathcal{B}^I}(J^{II}) \geq \\ &\geq \mathsf{opt}_{\mathcal{B}_{i-1}}(J) - p(S_{i-1}) - \mathsf{opt}_{\mathcal{B}^I}(J^{II}).\end{aligned}$$

From which we get:

$$p(S^I) + \mathsf{opt}_{\mathcal{B}^{II}}(J^{II}) \geq \mathsf{opt}_{\mathcal{B}_{i-1}}(J) - (p(S_{i-1}) - p(S_i)) - \mathsf{opt}_{\mathcal{B}^I}(J^{II}). \quad (1)$$

We bound each term in (1) separately. By Lemma 3, $\mathsf{opt}_{\mathcal{B}_{i-1}}(J) \geq (1 - P/k)\mathsf{opt}(J)$. Since Termination Condition 2 applied, $p(S_i) \geq (1 - 1/k)p(S_{i-1})$, and $(p(S_{i-1}) - p(S_i)) \leq p(S_{i-1})/k \leq \mathsf{opt}(J)/k$.

To bound $\mathsf{opt}_{\mathcal{B}^I}(J^{II})$, note that $\mathcal{B}^I$ is non-empty only if PartitionTimeLine was terminated due to Termination Condition 2. In that case, $J^{II} = J \setminus S_{i-1} = \bigcup_{j=1}^{r-1}(S_{j-1} \setminus S_j)$, where r is the number of iterations performed by PartitionTimeLine. Since each block in $\mathcal{B}^I$ is contained within blocks produced in each iteration, it follows from Lemma 4 that the profit of jobs from J^{II} scheduled by OPT(J) in each block $B \in \mathcal{B}^I$ is at most $\sum_{i=1}^{r-1} \alpha k^{i+2} \leq 2\alpha k^{r+1}$. From the operation of PartitionTimeLine, jobs with total profit at least k^{r+2} from S^I were scheduled in each block $B \in \mathcal{B}^I$. Therefore, $\mathsf{opt}_{\mathcal{B}^I}(J^{II}) \leq \frac{2\alpha}{k} p(S^I) \leq \frac{2\alpha}{k}\mathsf{opt}(J)$.

Substituting these bounds into (1) gives:

$$\begin{aligned}p(S^I) + \mathsf{opt}_{\mathcal{B}^{II}}(J^{II}) &\geq (1 - P/k)\mathsf{opt}(J) - \mathsf{opt}(J)/k - 2\alpha \cdot \mathsf{opt}(J)/k \\ &= (1 - \varepsilon)\mathsf{opt}(J)\ .\end{aligned}$$

Lemma 6. *In each block $B \in \mathcal{B}^{II}$,* OPT(J) *schedules at most $2\alpha \cdot k^{k \ln k + 3}$ jobs from J^{II}.*

Proof. Each job in J^{II} was removed from the schedule during one of the iterations. Thus, $J^{II} = \bigcup_{i=1}^{r}(S_{i-1} \setminus S_i)$, where r is the number of iterations performed by PartitionTimeLine. Since each block in $\mathcal{B}^I$ is contained within blocks produced in each interation, it follows from Lemma 4 that the total profits of jobs from J^{II} scheduled by OPT(J) in each block $B \in \mathcal{B}^{II}$ is at most $\sum_{i=1}^{r} \alpha k^{i+2} \leq 2\alpha k^{r+2} \leq 2\alpha k^{klnk+3}$ (recall that the maximum number of iterations is $\lceil k \ln k \rceil$).

To complete the proof of Lemma 1 it remains to show that PartitionTimeLine runs in time $O(n \cdot Q(n,m) \cdot \ln(P/\varepsilon))$. This is so since the loop has at most $\lceil k \ln k \rceil$ iterations, and in each iteration the dominating time is $O(Q(n,m)|\mathcal{B}|)$ for executing SC at most $|\mathcal{B}|$ times, and $|\mathcal{B}| \leq n\varepsilon$.

The proof of Lemma 1 is complete.

2.2 Proof of Lemma 2

The first step for proving Lemma 2 is defining a linear program whose integer solutions are feasible schedules. Then, we solve the program, and use randomized

rounding to obtain a feasible schedule with expected approximation ratio $1 - 1/e$. Finally, the algorithm is derandomized using the method of conditional expectations.

Let $\mathcal{S} = \mathcal{S}(J, \mathcal{I})$ be the set of all sequences $(j_1, \ldots, j_t, I_1, \ldots, I_t)$ of t distinct jobs in J and t (not necessarily distinct) intervals in $\mathcal{I}$, $0 \le t \le q$. Clearly, $|\mathcal{S}| < \sum_{i=1}^{q} n^i m^i \le 2(nm)^q$. Given $B \in \mathcal{B}$, any (feasible) schedule S of $t \le q$ jobs within B defines a unique sequence $(j_1, \ldots, j_t, I_1, \ldots, I_t) \in \mathcal{S}$, where each j_i is processed in I_i and after j_{i-1}. Given such a sequence we can find a feasible schedule within B defining it or determine that such does not exist in time $O(t) = O(q)$ time, by attempting to schedule every job within the corresponding interval in the sequence. This attempt is done by taking each job from the sequence in turn, and placing it within B as early as possible given the job's interval and avoiding overlap with jobs already scheduled during this attempt. Clearly, any two schedules within B defining the same sequences have the same profits, thus we identify any feasible schedule S within B with the sequence it defines. The profit $p_{(S,B)}$ of $(S, B) \in \mathcal{S} \times \mathcal{B}$ is the profit of some schedule that S defines within B, if such schedule exists, and $p_{(S,B)} = 0$ otherwise (that is, if no schedule within B defining S exists). Let $\mathcal{R} = \{(S, B) \in \mathcal{S} \times \mathcal{B} : p_{(S,B)} > 0\}$ and let $\mathcal{S}_B = \{S \in \mathcal{S} : p_{(S,B)} > 0\}$. For $(S, B) \in \mathcal{R}$ and $j_i \in S$ let $p_{(S,B)}(j_i) = p_{j_i, I_i}$. For every $(S, B) \in \mathcal{R}$ introduce a variable $y_{(S,B)}$ which may be interpreted as the "amount of S selected in the block B". Then integer feasible solutions to the following linear program correspond to feasible schedules within the blocks of $\mathcal{B}$.

$$\begin{aligned} \max \ & \textstyle\sum_{(S,B)\in\mathcal{R}} y_{(S,B)} \cdot p_{(S,B)} && && (2)\\ \text{s.t.} \ & \textstyle\sum_{(S,B)\in\mathcal{R}, j\in S} y_{(S,B)} \le 1 && \forall j \in J \\ & \textstyle\sum_{S\in\mathcal{S}_B} y_{(S,B)} = 1 && \forall B \in \mathcal{B} \\ & y_{(S,B)} \ge 0 && \forall (S,B) \in \mathcal{R} \end{aligned}$$

Note that this LP has $|\mathcal{R}| \le 2|\mathcal{B}|(nm)^q$ variables and $n + |\mathcal{B}|$ constraints (that are not just non-negativity constraints). We will apply a standard randomized rounding on y to obtain an integral feasible solution $\tilde{y}$ to (2). The rounding procedure is as follows.

1. For each block B choose randomly with distribution $y_{(S,B)}$ a unique schedule S_B assigned to block B at this stage (possibly $S_B = \emptyset$).
2. For every job j assigned to more than one block, remove j from all schedules containing it, except from one that has maximum $p_{(S,B)}(j)$.

Let $\tilde{y}$ be an integral solution derived from y by such randomized rounding, and let $\tilde{\nu} = \sum_{(S,B)\in\mathcal{R}} \tilde{y}_{(S,B)} \cdot p_{(S,B)}$ be the (random variable corresponding to the) profit of the schedule specified by $\tilde{y}$, and let ν be the optimal value of (2). We will prove that the expected value of $\tilde{\nu}$ is at least $(1 - 1/e)\nu$. The proof is similar to the proof of [6, Theorem 2.1] where MAX-GAP was considered and is presented here only for completeness of exposition. We use the following statement from [6]:

Lemma 7 ([6], Lemma 2.1). *Let $y_1, \ldots, y_\ell$ be a sequence of non-negative reals so that $\sum_{i=1}^{\ell} y_i \leq 1$ and let $p_1 \geq p_2 \geq \cdots \geq p_\ell \geq 0$. Then*

$$p_1 y_1 + p_2(1-y_1)y_2 + \cdots + p_\ell \left[\prod_{i=1}^{t-1}(1-y_i)\right] y_\ell \geq (1-(1-1/\ell)^\ell) \sum_{i=1}^{\ell} p_i y_i \ .$$

Lemma 8. *The expected value of $\tilde{\nu}$ is at least $(1-1/e)\nu$.*

Proof. Let $j \in J$. Sort the profits $p_{(S,B)}(j)$, $(S,B) \in \mathcal{R}$, in a decreasing order

$$p_{(S_1,B_1)}(j) \geq p_{(S_2,B_2)}(j) \geq \cdots \geq p_{(S_\ell,B_\ell)}(j) \ .$$

For simplicity, denote $p_i = p_{(S_1,B_1)}(j)$ and $y_i = y_{(S_i,B_i)}$. Let $\nu(j) = \sum_{i=1}^{\ell} p_i y_i$ be the "profit of j" in LP (2), and note that $\nu = \sum_{j \in J} \nu(j)$. Let $\tilde{\nu}(j)$ be the (random variable corresponding to the) profit from job j in the schedule computed by the algorithm. By the linearity of expectation, it would be sufficient to prove that for every $j \in J$ the expected value of $\tilde{\nu}(j)$ is at least $(1-1/e)\nu(j)$. This follows from Lemma 7, since the expected value of $\tilde{\nu}(j)$ is:

$$p_1 y_1 + p_2(1-y_1)y_2 + \cdots + p_\ell [\prod_{i=1}^{t-1}(1-y_i)] y_\ell \geq (1-(1-1/\ell)^\ell) \sum_{i=1}^{\ell} p_i y_i \geq (1-1/e)\nu(j) \ .$$

The algorithm can be derandomized using the method of conditional probabilities. We state the algorithm for the analysis of the time complexity, but omit the proof of its validity, as it is identical to the one in [8] where MAX-GAP was considered. Given $\mathcal{B}' \subseteq \mathcal{B}$ and $J' \subseteq J$ let $\nu(J', \mathcal{B}')$ denote the optimal value of (2) with J replaced by J' and $\mathcal{B}$ by $\mathcal{B}'$. The algorithm is as follows.

Initialization: $J' \leftarrow J$, $\mathcal{B}' \leftarrow \mathcal{B}$.
For every $B \in \mathcal{B}$ do:
  Schedule in B a set $S_B \subseteq J'$ of jobs that maximizes $p_{(S,B)} + \nu(J' \setminus S, \mathcal{B}' \setminus \{B\})$;
  $J' \leftarrow J' \setminus S_B$, $\mathcal{B} \leftarrow \mathcal{B}' \setminus \{B\}$.
EndFor

To complete the proof of the Lemma 2 it remains to show the time complexity. The time required to compute the profits $p_{(S,B)}$ is $O(q|\mathcal{B}|(nm)^q)$, $O(q)$ time per variable. We assume that this time is dominated by the time required to solve the linear program (2), which is $O\left(L(|\mathcal{B}|(nm)^q, n+|\mathcal{B}|)\right)$. This is the time complexity of the randomized version. The time complexity of the deterministic version is $O(|\mathcal{B}|(nm)^q \cdot L(|\mathcal{B}|(nm)^q, n+|\mathcal{B}|))$, as claimed.

The proof of Lemma 2, and thus also of Theorem 1, is complete.

3 Proof of Theorem 2

We reduce the following NP-complete problem [7] to θ-SWI.

Subset-Sum
Instance: A set $A = \{a_1, \ldots, a_n\}$ of positive integers, and another integer B.
Question: Is there $A' \subseteq A$ such that the integers in A' sum to exactly B?

The reduction is as follows. Let $1 < \alpha < \min\{2, \theta\}$ be arbitrary. Let $K = B \cdot \lceil 1/(\alpha - 1) \rceil$, and let $T = nK + B$. For each $j = 1, \ldots, n$ there are two jobs j^- and j^+ each having the same single interval $I_j = [(j-1)K, jK + B)$, and processing time $\ell_{j^-} = K$ and $\ell_{j^+} = K + a_j$, respectively. The profit of each job equals to its processing time, and set $T = nK + B$.

Since $|I_j| = K + B$ and $\ell_{j^+}, \ell_{j^-} \geq K$ for each j, the stretch factor of the obtained instance of SWI is at most $(K + B)/K = 1 + \lceil 1/(\alpha - 1) \rceil^{-1} \leq \alpha < \theta$. Thus we obtain a θ-SWI instance.

Lemma 9. *The answer to* Subset-Sum *is* YES *if, and only if, the* SWI *instance can achieve a total profit of* T.

Proof.
The *if* part.
Let S be a feasible set of jobs that achieves a total profit of T. We claim that then for each j *exactly* one of j^-, j^+ is in S. To see this note that for each $1 \leq j \leq n$:

(i) at most one of j^-, j^+ is in S, since $|I_j| = K + B < 2K + a_j = \ell_{j^-} + \ell_{j^+}$;
(ii) at least one of j^-, j^+ is in S, since each time window I_j contains the non-empty sub-interval $[(j-1)K + B, jK)$ that is disjoint to all the other time windows.

For $j = 1, \ldots, n$ let b_j be the boolean variable with $b_j = 1$ if $j^+ \in S$ and $b_j = 0$ otherwise (that is, if $j^- \in S$). The total profit of the jobs in S is exactly $\sum_{j=1}^{n}(K + b_j a_j) = nK + \sum_{j=1}^{n} b_j a_j$. Comparing this to the total profit $T = nK + B$, we get $\sum_{j=1}^{n} b_j a_j = B$. This defines a solution to the Subset-Sum instance.

The *only if* part.
Let $A' \subseteq A$ such that A' sums to B. We will show a solution to the SWI instance that achieves a profit of T. Let $b_j = 1$ if $a_j \in A'$ and $b_j = 0$ otherwise. Create the following solution S to SWI. For each $j = 1, \ldots, n$ do: if $b_j = 1$ add the job j^+ to S, with the start time $(j-1)K + \sum_{i<j} b_i a_i$; otherwise, add the job j^- to S, with the same start time. It is easy to verify that this is a feasible solution to SWI with a total profit of T. □

The following dynamic-programming algorithm computes an optimal solution for 2-SWI. Define the *latest start* of job j as $t_j = d_j - \ell_j$. The jobs are sorted in order of non-decreasing t_j, breaking ties arbitrarily, so assume $t_1 \leq t_2 \leq \cdots \leq t_n$. For $1 \leq i \leq n$ and $0 \leq p \leq \sum_j p_j = \mathcal{P}$ define $D[i, p]$ as the minimal ending time of a feasible schedule $S \subseteq \{1, \ldots, i\}$ with total profit exactly p; $D[i, p] = \infty$ if no such feasible S exists. The optimal profit eventually equals to $\max\{p : D[n, p] < \infty\}$.

The table D can be computed using the following recurrence. Set $D[1,p] = r_1 + \ell_1$ if $p = p_1$ and $r_1 + \ell_1 \leq d_1$, and $D[1,p] = \infty$ otherwise. Let $2 \leq i \leq n$. If $p_{i+1} > p$ set $D[i,p] = D[i-1,p]$. For $p_{i+1} \leq p$ set

$$s_i = \max\{D[i-1, p-p_i], r_i\}\,.$$

If $s_i + \ell_i > d_i$ set $D[i,p] = D[i-1,p]$. Otherwise,

$$D[i,p] = \min\{s_i + \ell_i, D[i-1,p]\}.$$

Lemma 10. *The table D is computed correctly in $O(\min\{n\mathcal{P}, rn^r \log n\})$ time, where $P = \sum_j p_j$.*

Proof. Note that if $s_j < s_i$ are feasible start times of jobs i, j, then $t_j < t_i$. Indeed, for θ-SWI, $t_j < s_j + (\theta-1)\ell_j \leq s_i + (\theta-1)\ell_j \leq t_i + (\theta-1)\ell_j$. In particular, for $\theta = 2$ we have $t_j < t_i$. Clearly, every entry is computed using previous entries. Any feasible solution defines a sequence of jobs according to their starting times. By the above, any such sequence must be a subsequence of the latest start sequence. Therefore, the step of computing $D[i,p]$ from previous entries requires examining just two options: whether or not to append job i to the constructed schedule. The decision is made by choosing the earliest-finishing one from these two options. The initial sorting of jobs requires $O(n \log n)$ time. The number of entries in the table D is $O(n\mathcal{P})$, and each entry is computed in constant time. The time bound $O(n \log n + n\mathcal{P}) = O(n\mathcal{P})$ follows.

For the $O(rn^r \log n)$ time bound, we compute all possible profits $\Pi = \{p(S) : S \subseteq J\}$ of subsets of the items (assuming any set of items can be chosen) and sort these profits in an increasing order. This can be done in $O(rn^r \log n)$ time, as follows. Let $p_1, \ldots p_r$ be the possible distinct item profits, and let n_i be the number of items in J of profit p_i. Then $|\Pi| \leq (n_1+1)(n_2+1)\cdots(n_r+1) \leq (n/(r+1))^r = O(n^r)$, and Π can be computed in $O(rn^r)$ time. Thus the total time required including sorting is $O(rn^r + n^r \log(n^r)) = O(rn^r \log n)$. Then we use the same dynamic programming algorithm, but define and fill in the table only the relevant entries. The number of entries in the table is $O(n^r)$, and each entry is computed in constant time.

The proof of Theorem 2 is complete.

Remark: A standard profit-truncation algorithm may be utilized to give a FPTAS for 2-SWI with running time $O(n^2/\varepsilon)$, which is the same as the one of [3]. This FPTAS is based on algorithm pseudo-polynomial in $\mathcal{P}$, while the one of [3] is based on an algorithm pseudopolynomial in $T = \max_{j \in J} d_j$. It remains open whether θ-SWI is APX-hard for any $\theta > 2$.

4 Conclusions and Open Problems

In this paper we gave a $(1 - 1/e - \varepsilon)$-approximation scheme for GSWI with bounded profits, while showing that SWI with small stretch factor are NP-hard. One open problem is obtaining a ratio better than $1/2$ for arbitrary profits. Another open problem is whether SWI with $\{0,1\}$-profits (or JISP) admits a better ratio than $(1 - 1/e)$.

Acknowledgment. We thank anonymous referees for useful comments.

References

1. Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., Shieber, B.: A unified approach to approximating resource allocation and scheduling. J. of the ACM 48(5), 1069–1090 (2001)
2. Bar-Noy, A., Guha, S., Naor, S., Schieber, B.: Approximating the throughput of multiple machines in real-time scheduling. SIAM J.Comput. 31(2), 331–352 (2001)
3. Berman, P., DasGupta, B.: Multi-phase algorithms for throughput maximization for real-time scheduling. J. Comb. Optim. 4(3), 307–323 (2000)
4. Chuzhoy, J., Ostrovski, R., Rabani, Y.: FOCS, pp. 348–356 (2001)
5. Feige, U., Vondrák, J.: Approximation algorithms for allocation problems: Improving the factor of $1 - 1/e$. In: FOCS, pp. 667–676 (2006)
6. Fleischer, L., Goemans, M.X., Mirrokni, V.S., Sviridenko, M.: Tight approximation algorithms for maximum general assignment problems. In: SODA, pp. 611–620 (2006)
7. Garey, M.R., Johnson, D.S.: W. H. Freeman, San-Francisco (1979)
8. Nutov, Z., Beniaminy, I., Yuster, R.: A $(1 - 1/e)$-approximation algorithm for the generalized assignment problem. Oper. Res. Lett. 34(3), 283–288 (2006)

Pricing Tree Access Networks with Connected Backbones

Vineet Goyal[1,⋆], Anupam Gupta[2], Stefano Leonardi[3,⋆⋆], and R. Ravi[4,⋆⋆⋆]

[1] Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213, USA
vineet@cmu.edu
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA
anupamg@andrew.cmu.edu
[3] Dipartimento di Informatica e Sistemistica, University of Rome "La Sapienza",
Via Salaria 113, 00198 Rome, Italy
leon@dis.uniroma1.it
[4] Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213, USA
ravi@cmu.edu

Abstract. Consider the following *network subscription pricing* problem. We are given a graph $G = (V, E)$ with a root r, and potential customers are companies headquartered at r with locations at a subset of nodes. Every customer requires a network connecting its locations to r. The network provider can build this network with a combination of *backbone* edges (consisting of high capacity cables) that can route any subset of the customers, and *access* edges that can route only a single customer's traffic. The backbone edges cost M times that of the access edges. Our goal is to devise a *group-strategyproof* pricing mechanism for the network provider, i.e., one in which truth-telling is the optimal strategy for the customers, even in the presence of coalitions. We give a pricing mechanism that is 2-competitive and $O(1)$-budget-balanced.

As a means to obtaining this pricing mechanism, we present the first primal-dual 8-approximation algorithm for this problem. Since the two-stage Stochastic Steiner tree problem can be reduced to the underlying network design, we get a primal-dual algorithm for the stochastic problem as well. Finally, as a byproduct of our techniques, we also provide bounds on the inefficiency of our mechanism.

1 Introduction

Consider the following connected backbone for tree access network (CBTAN) design problem: given an undirected graph $G = (V, E)$ with metric costs $c(e)$ on the edges, and a root r, we want to build a network to connect a set of l possible *customers* U. The i^{th} customer is specified by a set $S_i \subseteq V$ of terminals. A solution to the problem is a set of connected backbone edges E_0 containing the root r and a set of access networks E_i

⋆ Supported in part by NSF ITR grant CCR-0122581 (The ALADDIN project) and NSF grant CCF-0430751.

⋆⋆ Part of this work was done while visiting the School of Computer Science at Carnegie Mellon University.

⋆⋆⋆ Corresponding author.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 498–509, 2007.

one for each customer i such that $E_0 \cup E_i$ contains a Steiner tree connecting $S_i \cup \{r\}$ for all i. Backbone edges E_0 are a factor M costlier than the access edges E_i. The total cost to connect any subset $U' \subseteq U$ of customers is $Mc(E_0) + \sum_{i \in U'} c(E_i)$. Note that the objective of minimizing the total network cost naturally trades off backbone and access network costs.

CBTAN is equivalent to the two-stage stochastic Steiner tree problem (StocST) studied in [IKMM04, GPRS04, GRS04], where the customers correspond to *scenarios* and the backbone network corresponds to the first-stage tree. The key to the equivalence is the fact that in StocST, an edge bought in the second stage in scenario s is weighted by $p_s \cdot \sigma$, where p_s is the probability of scenario s and σ is the inflation factor in the second stage. By setting $M = \frac{1}{p_s \cdot \sigma}$, we can infer the equivalence. In line with this analogy, we refer to customers as *scenarios*. Also, we refer to the nodes connected by the backbone edges E_0 to the root r as *backbone nodes* or *facilities*. Our problem generalizes the problem of network design for information networks defined by Hayrapetyan et al. [HST05] by imposing connection between facilities. The single-commodity rent-or-buy (SROB) network design problem [SK04, GKR03, PT03] can also be derived from our problem if every scenario is a single terminal.

In this paper, we are interested in a *game-theoretic extension* of the problem that is perhaps best described as a *subscription pricing* problem: suppose we have a set U of l customers, where customer i's ultimate goal is to connect the set S_i to the root r, and she derives a (privately-held) *utility* $u_i \geq 0$ from being connected to the root. We can sell *subscriptions* to the potential customers such that a subscription-holding customer is guaranteed connectivity of her set to the root. The goal is to price these subscriptions for potential customers in such a way that the sale of these subscriptions to some subset of customers yields enough money (up to constant factors) to pay for the cost of the network serving these subscription-holding customers. Note that the backbone edges built by the algorithm depends on the subset of customers accepted.

Formally, we are interested in finding a *cost-sharing mechanism* that determines *group-strategyproof* subscription prices ξ_i for each customer i in U. A mechanism is group-strategyproof if reporting one's true utility u_i as the bid is a dominant strategy for each customer, and the customers should have no incentive to indulge in strategic behavior *even when they are allowed to collude*. The mechanism solicits bids $\{b_i\}_{i \in U}$ from all customers and commits to serve the customers in U iff $\xi_i \leq b_i, i \in U$. If there is a customer whose bid is less than her subscription price (as determined by the mechanism), the process is repeated after removing all the customers whose bid is lower than their subscription price.

Of course, making the subscriptions free would ensure truthfulness; to avoid such degenerate solutions, we would like to ensure other desirable properties. E.g., a mechanism is *budget balanced* if the actual cost $C(S)$ of servicing the customers in S is at most the sum of the subscription costs for the customers in S—i.e., we recoup our costs by selling the subscriptions. (In this paper, we focus on α-budget balance, where we only recover a $1/\alpha$ fraction of $C(S)$; hence budget-balance is the same as 1-budget-balance.) A mechanism is *β-competitive* if the sum of subscription costs to the customers in S does not exceed β times the cost of an optimal solution for S. A mechanism is called *efficient* if it selects a set S of customers that maximizes the

efficiency $u(S) - C(S)$. A mechanism that selects an empty set of customers and returns an empty solution is both budget-balanced and competitive but not efficient. Thus, efficiency is an important property for a mechanism.

Classical results in economics [GKL76, Rob79] state that budget balance and efficiency cannot be simultaneously achieved by any mechanism. Moreover, Feigenbaum et al. [FPS01] recently showed that there is no group-strategyproof mechanism that always recovers a constant fraction of the maximum efficiency and a constant fraction of the incurred cost even for the simple fixed-tree multicast problem. In light of these impossibility results, previous work on mechanism design usually focused on a proper subset of the above desiderata. One class of such mechanisms are based on a framework of Moulin and Shenker [MS01], who show that given an α-budget balanced and *cross-monotone cost sharing method* for the underlying problem, the natural *Moulin mechanism* [Mou99] satisfies both α-budget balance and group-strategyproofness. Cross-monotoneity imposes that the cost-share computed by the mechanism for each player only decreases if more players join the game. (Formal definitions are deferred to Section 2.) Moulin and Shenker's framework has recently been applied to game-theoretic variants of numerous classical optimization problems, and we will also seek cross-monotone cost-shares for our network design problem to solve the subscription pricing problem.

Contributions. Our main result is the following:

Theorem 1.1. *There is a cross-monotone cost sharing scheme for* CBTAN *that is 2-competitive and 41-budget-balanced.*

As in several previous papers giving cost-shares, we derive these cost shares from a primal-dual algorithm for the CBTAN problem. We obtain the first primal-dual algorithm for CBTAN and StocST that achieves an 8-approximation for these problems. Due to lack of space, we defer the description of the algorithm for the full version of the paper.

The duals generated in such a primal-dual algorithm naturally give us cost-shares that are competitive and approximately budget-balanced. However, they are not cross-monotone and hence, we have to work harder to achieve this property. We are able to extend the results of Pál and Tardos on SROB network design [PT03] to obtain cross-monotone cost shares for this more general setting; the details of this process (and hence the proof of the above theorem) appear in Section 3.

Roughgarden and Sundararajan [RS06] introduced an alternative measure of efficiency that can be approximated at the same time of budget balance. In Section 4, we present results on the approximated efficiency achieved by our mechanism.

Related Work. Approximation algorithms for a variant of the CBTAN problem where the access network involves direct edges to the backbone nodes have been well-studied in [RS99, GKK+01, SK04, KM00]. Other variants where no connectivity is sought among the backbone nodes are studied in [AZ02].

The Stochastic Steiner Tree problem [IKMM04, GPRS04, GRS04] is equivalent to CBTAN and constant-factor approximations based on randomized selection strategies are known; however, no primal-dual algorithms were known for the problem.

See Moulin and Shenker [MS01] for a study of group-strategyproof mechanisms and how to use cross-monotone cost sharing methods to design such mechanisms using the Moulin mechanism [Mou99]. This work has given game-theoretic variants of problems like fixed-tree multicast [AFK+04, FKSS03, FPS01], facility location [PT03], submodular cost-sharing [MS01], Steiner trees [JV01, KSK96], single-source rent-or-buy network design [PT03, LS04, GST04], and Steiner forests [KLS05]. Lower bounds on the budget balance that can be achieved by cross-monotone cost shares, are given in [IMM05, KLSvZ05].

2 Preliminaries on Cost Sharing Methods

A *cost sharing method* ξ for a problem Π is an algorithm that, given any subset $S \subseteq U$ of players demanding service, computes a solution for the set S; moreover, it computes a non-negative cost share $\xi_i(S)$ for each player $i \in S$. The following properties of cost-sharing methods will be useful.

Definition 2.1. *We say that the cost sharing method ξ is* β-budget balanced *if for every subset $S \subseteq U$,*

$$\frac{1}{\beta} \cdot C(S) \leq \sum_{i \in S} \xi_i(S) \leq C(S).$$

A cost sharing method ξ is cross-monotone *[MS01] if for any two sets S and T such that $S \subseteq T$ and any player $i \in S$ we have $\xi_i(S) \geq \xi_i(T)$; i.e., the cost shares of a player never increase if more players enter the game.*

Moulin and Shenker [MS01] showed that, given a cost sharing method ξ that is budget balanced and cross-monotone for the underlying problem, the following cost sharing mechanism $M(\xi)$ (henceforth known as the *Moulin mechanism*) satisfies budget-balance and group-strategyproofness: *initially, let $S \rightarrow U$. If for each player $i \in S$ the cost share $\xi_i(S)$ is at most her bid b_i, we stop. Otherwise, remove from S all players whose cost shares are larger than their bids, and repeat*. Eventually, let $\xi_i(S)$ be the costs that are charged to players in the final set S.

3 A Cross-Monotone Cost-Sharing Scheme for CBTAN

In this section, we develop a cross-monotone cost-sharing method that is competitive and budget balanced for CBTAN. The algorithm in this section can be perceived as a substantial extension of the primal-dual algorithm, where instead of running one primal-dual process, we run *an extra dual process* called, as in [PT03], the *ghost process*—this is a monotone process used to generate the cost shares; the heart of the argument is in relating the real and ghost processes to each other and arguing that the cost shares generated by the ghost process are enough to pay for the actual network created.

3.1 The Real and Ghost Processes

Note that if the we fix the set of connected backbone edges E_0, we can obtain a good approximation to the access edges for each customer or scenario (say, by using a MST

heuristic to connect to the backbone) to complete the solution. Thus, the problem essentially reduces to finding a low cost set of backbone edges such that there is a low cost of completion (set of access edges) for each scenario. Recall that the nodes connected to the root by the backbone edges are referred to as facilities or backbone nodes. Finding the set of backbone edges is equivalent to finding the set of facilities that are connected to the root by the backbone edges. The access edges of each scenario will form a Steiner forest on the terminals of the scenario, each tree of the forest containing at least one backbone node. Thus, we would consider the CBTAN problem as one of finding facilities that are connected to the root through the backbone edges.

We describe a *ghost process*, which is similar to the dual ascent schema of [AKR95], to construct the solution and cost shares for all the scenarios. It is similar in spirit to the idea of ghost process developed by Pál and Tardos in [PT03] for SROB, where the ghost of each terminal j is a ball with center j and growing uniformly to infinity. However, unlike the mechanism for SROB where each scenario terminal has a direct connection to some open facility, our ghost process has to assign cost share for building Steiner tree connections to open facilities. This is done by integrating the ghost process with l separate dual ascent Steiner forest processes [AKR95]. For simplicity, we maintain $l+1$ different copies $(G_0, G_1, \ldots, G_l)$ of the graph G. Copies $G_1, \ldots, G_l$ correspond to the l scenarios and copy G_0 corresponds to the open facilities. Initially, every singleton terminal of scenario i is an active component in the copy G_i.

During the course of the algorithm, we would open some locations in G_0 as *tentative facilities* after M or more dual ascent processes in the other copies have made the location tight (in their respective dual packing constraints). Such a location is a feasible location to open a facility as it has clustered M different scenario demands. For simplicity, we assume that a facility can be opened anywhere along an edge. We can easily remove this assumption at no additional cost.

We open a *real facility* at a tentative location j only if there is no real open facility within a distance $4t_j$ from j, where t_j is the time when j was declared tentatively open. We define a corresponding ghost process in copy G_0 of the graph, where we run a dual ascent process on tentatively open facilities. Each tentatively open facility p becomes an active component in G_0 at the instant it is declared open, say t_p.

Definition 3.1 (Tentative Facility Moats). *We call the components in G_0, the* tentative facility moats*. The ghost of a tentatively open facility p opened at time t_p is defined for any time $t \geq t_p$ as a ball $\mathcal{B}(p, t - t_p)$ of radius $t - t_p$ around vertex p. Tentative facility moats in G_0 are therefore the union of ghost components of different radii.*

Definition 3.2 ((Ghost) Scenario Moats). *In each scenario graph G_i, at any time t, we define a collection of active subsets also called* ghost scenario moats *or just* scenario moats*. Every terminal in each scenario starts a ghost moat which includes all nodes in a ball $\mathcal{B}(v, t)$ of radius t around vertex v at time $t \geq 0$. As time progresses, such a ghost moat can eventually collide with (i) either another ghost of scenario i (in G_i), or (ii) a tentatively facility moat (in G_0), to merge into a single active ghost moat. The set of scenario moats of scenario i at time t is the set of disjoint active subsets of scenario i in G_i.*

Definition 3.3 (Dark and Lit Moats). *We call a scenario moat* dark *if it does not contain any open facility (tentative or real) and* lit *if it contains at least one tentatively or real open facility.*

Initially all the scenario moats are dark. The ghost process results in one of the following events:

Events in the Ghost Process

1. Two dark scenario moats C and C' intersect in some copy G_i of the graph. The two moats merge to form a new dark scenario moat $C \cup C'$.
2. For some location $j \in V$, at least M scenario moats of different scenarios (i.e. M moats in different copies of the graph) contain j for the first time.
   (a) Declare j as a tentatively open facility. The singleton terminal j becomes an active component in G_0.
   (b) All the scenario moats containing j are declared "lit".
   (c) If there is no real open facility within a distance $4t_j$ (t_j is the current time) from j, then open a real facility at j.
3. A lit scenario moat C intersects a dark or a lit moat C' in some copy G_i of the graph. The two moats merge to form a new lit moat.
4. A scenario moat C (dark or lit) of some scenario $i \neq 0$ intersects a tentative facility moat $\mathcal{F}$ in G_0. Declare the scenario moat C lit if it was dark and merge C with $\mathcal{F}$. Thus, the new lit moat in G_i is $C \cup \mathcal{F}$.

We continue this ghost process until every scenario moat contains the root. The ghost process described above lets us decide the cost shares for each terminal and also determines where to open real facilities.

Network Design Algorithm

- Build a Steiner forest E_i for each scenario i, that connects terminals in scenario i to closest real facilities (for each component).
- Build a Steiner tree (of backbone edges) over the real facilities connecting them to the root.

3.2 Defining the Cost Shares

We now describe the cost shares that are collected by terminals of all the scenarios during the ghost process. We assign two kinds of cost shares to every terminal: **(a)** one when it is a part of a dark scenario moat, and **(b)** another when it becomes a part of a lit scenario moat. Let us define the two cost shares for a terminal j in scenario i. Let $C_j(t)$ be the scenario moat containing j at time t and t_j^1 be the first time instant when j is contained in a lit scenario moat and let t_j^2 be the time when the moat containing j reaches the root. Thus, the cost share for j till t_j^1 is defined as:

$$f_j^1 = \int_{t=0}^{t_j^1} \frac{1}{|C_j(t)|} dt$$

Here $|C_j(t)|$ denotes the number of terminals in the scenario moat $C_j(t)$ that divide up the cost share accumulated as dual by this growing moat. For $t \geq t_j^1$, j is in a lit moat $C_j(t)$.

Definition 3.4. *We say that the moat $C_j(t)$ contributes to a tentative facility moat $\mathcal{M}$ if there exists a terminal $k \in C_j(t)$ which is at a distance at most t from the moat $\mathcal{M}$.*

Note that $C_j(t)$ could possibly contribute to many facility moats. Suppose $C_j(t)$ contributes to moats $\mathcal{M}_1, \ldots, \mathcal{M}_l$ and let $n(\mathcal{M}_i)$ be the number of different scenarios contributing to moat $\mathcal{M}_i$ at time t. Also, let $n_{C_j(t)} = \max_{i=1,2,\ldots,l} n(\mathcal{M}_i)$. The cost share for the terminal j is:

$$f_j^2 = \int_{t=t_j^1}^{t_j^2} \frac{M}{|C_j(t)| \cdot n_{C_j(t)}} dt$$

3.3 Properties of the Cost Shares

We need to prove that the cost shares defined above are competitive, cross-monotone and budget balanced. To prove competitiveness, we construct a feasible dual for the CBTAN problem from the cost shares of the terminals. Since a feasible dual is a lower bound on the optimum cost, it proves that cost shares are competitive (approximately). The cross-monotoneity property follows from the description of the ghost process. The crucial part is proving that cost shares are budget-balanced. In other words, the cost shares of the terminals can pay for the cost of the network constructed by our algorithm. Charging the cost of access networks (Steiner forest E_i) for each scenario to the cost shares collected by the terminals of that scenario is not very difficult and follows standard primal-dual arguments [AKR95]. However, proving that the total cost shares of all terminals are sufficient to pay for the Steiner tree over the real facilities is challenging and requires new ideas and charging techniques. In the following lemmas, we will prove the desired properties for the cost shares. Due to lack of space, some of the proofs have been deferred to the full version.

Lemma 3.1. *The cost shares ($f_j^1 + f_j^2$ of terminal j) defined by the dual ascent process are 2-competitive i.e. $\sum_{k=1}^{l} \sum_{j \in S_k} (f_j^1 + f_j^2) \leq 2OPT$, where OPT is the optimal cost for the CBTAN problem.*

Proof. It is sufficient to prove that the total cost shares of all the terminals is at most two times the optimal solution. We will show that half times the cost shares form a feasible dual. Consider a moat C at time t of scenario i. If C is a dark scenario moat at time t, the dual $\beta_{C,i}$ increases at a rate half, i.e. $\frac{d}{dt}\beta_{C,i} = \frac{1}{2}$. If C is a lit scenario moat at time t, then $\frac{d}{dt}\beta_{C,i} = \frac{M}{2n_{C(t)}}$. Here, we assume that there are locations at each point along every edge and dt is an infinitesimal amount of time. Clearly, the individual scenario constraint for edge packing is never violated. Consider the dual constraint that bounds the cumulative packing of edge e by $M \cdot c_e$. When an edge e is dark, i.e. no tentative facility has been opened on any location on e, each scenario moat, that e crosses, collects cost share at a rate 1. Thus, the total dual collected by moats which e crosses during the time it was dark is at most $\frac{(M-1)c_e}{2}$, because at most $M-1$ scenarios components can cross e while it is dark. When it becomes lit, the total dual collected by all moats that e crosses after this instant of time is at most $\frac{Mc_e}{2}$. Thus, the above constraint is not violated by the scaled dual. Thus, we have that $\sum_{j \in V} \frac{f_j^1 + f_j^2}{2} \leq \text{OPT}$ or that $\sum_{j \in V} (f_j^1 + f_j^2) \leq 2\text{OPT}$.

Lemma 3.2. *The cost share $f_j^1 + f_j^2$ for any terminal j is cross-monotone.*

Budget balance. The proof of budget balance proceeds in two parts. In the first part, we prove that the cost shares f^1 of terminals are enough to connect terminals of a scenario to a real open facility. This is proved via a standard argument in the following lemma.

Lemma 3.3. *For any scenario i, we can build a Steiner forest over terminals in S_i such that each Steiner component is connected to some open facility and the cost of the Steiner forest is at most $8 \sum_{j \in S_i} f_j^1$.*

In the second part, we prove that the cost shares can pay for building a Steiner tree over the open facilities. This is the more difficult part of the proof and is proved over the following series of lemmas. For the sake of the analysis, we consider the Steiner tree algorithm over real facilities being run in parallel to the ghost process.

Note that after a scenario moat becomes lit, it collects cost share at a rate that is less than 1. This may not be sufficient to pay for Steiner connections between real facility moats, whose cost is M times the cost of the connection. In this case, however, we argue that the cost share collected by the scenario moats at a time $t' \leq t$ is sufficient to pay for the share requested by real facility moats at time $5t$.

We charge the cost of the Steiner connections between real facility moats to a *merge tree* over the dark and the lit moats of each scenario. The merge tree is a virtual tree which we construct during the ghost process. Each edge e in the merge tree has an association fraction $f(e)$ which is decided during the ghost process. $f(e)$ is the fraction of the cost of e which can be paid by the cost shares of the terminals within a constant factor.

Merge Tree. To construct the merge tree for scenario i, we consider a slightly modified view of the ghost process in copy G_i of the graph corresponding to scenario i. Suppose a lit moat $\mathcal{M}$ intersects with a tentative facility moat F at time t in the ghost process. Recall that we merge $\mathcal{M}$ and F to form a new lit moat $\mathcal{M}$ in the ghost process. In the modified view, we call the tentative facility moat F at time t, a *hole* H in G_i.

Claim. Consider a moat $\mathcal{M}$ in scenario i at time t. Any location $j \in \mathcal{M}$ is at a distance of at most t from a scenario terminal $v \in \mathcal{M}$ or a hole $H \subset \mathcal{M}$ created during the ghost process.

The merge tree for scenario i is constructed as follows:

1. Suppose a moat $\mathcal{M}_1$ merges with a tentative facility moat F at time t at location j. There exists a terminal $v_1 \in \mathcal{M}_1$ or a hole $H_1 \subset \mathcal{M}_1$ which is at a distance t from j (wlog say v_1). In the merge tree $MT(i)$, we construct an edge e between v_1 and j. The fraction $f(e)$ associated with e is the rate at which $\mathcal{M}_1$ collects cost share at time t.
2. Suppose two moats $\mathcal{M}_1$ and $\mathcal{M}_2$ of scenario i merge at location j at time t. We can assume wlog that j is at a distance t from some terminal $v_1 \in \mathcal{M}_1$ and some hole $H_2 \subset \mathcal{M}_2$. In the merge tree $MT(i)$ we construct an edge e between v_1 and closest location $h \in H_2$. The fraction $f(e)$ associated with the edge is the maximum of the rates at which $\mathcal{M}_1$ and $\mathcal{M}_2$ collect cost shares at time t.

Lemma 3.4. *The total cost share collected by the terminals of a scenario i is at least a fraction $1/4$ of the total cost of $MT(i)$.*

Recall that the dual ascent process for Steiner tree on the real open facilities continues in assumed to run in parallel to the ghost process. The following notation will be used in the remainder of the proof.

Definition 3.5. *The following components will be crucial to the following discussion:*

- ***Ghost component:*** *A tentative facility moat at time t and all the terminals of different scenarios which are within distance t of the moat.*
- ***Set of contributors*** *of real facility moat $\mathcal{M}_t$ at time t: set of scenario terminals which are within a distance of $\max\{t, t_p\}$ from a real facility p in moat $\mathcal{M}_t$ (where t_p is the opening time of facility p).*
- ***Real component:*** *a real facility moat $\mathcal{M}_t$ at time t and its set of contributors.*

The following lemma is a natural consequence of the condition for opening real facilities.

Lemma 3.5. *Any scenario terminal v is contained in one real component at any time.*

The next lemma, similar in spirit to the one in [PT03], helps in relating the cost shares collected by the set of contributors to the cost of the Steiner tree over open facilities.

Lemma 3.6. *The set of terminals contained in a ghost component at time t will be contained in the same real component at time $5t$.*

We can now prove that the cost shares collected by the scenario moats are enough to pay for the Steiner tree over real facilities. Recall that for any real open facility p there is no other real open facility within a distance $4t_p$ from p, where t_p is the time when p is declared open in the ghost process. Let $\mathcal{M}_p$ denote the real component containing p. Until time t_p, facility p is the only open facility in $\mathcal{M}_p$. So, the terminals within a distance of t_p from p form the contributor set for $\mathcal{M}_p$ at any time $t \leq t_p$. The following lemma proves that we can charge the cost of growing real components in the time interval $[0, t_p]$ to the first-stage cost shares of the set of contributors of the real component containing facility p at time t_p.

Lemma 3.7. *Consider a real open facility p and suppose $\mathcal{M}_p$ is the real component containing p till time t_p. The cost shares collected by the set of contributors of moat $\mathcal{M}_p$ can pay for its growth till time t_p.*

The following lemma proves that the cost shares can continue to pay for the growth of real facility moats after the time facilities got opened in the ghost process.

Lemma 3.8. *Consider a real component R at time $5t$. Suppose $5t > t_p$ for all real facilities $p \in R$. The demanding rate of R at time $5t$ can be satisfied by the cost shares collected by its set of contributors at some time $t' \leq t$.*

Proof. Consider a real component R at time $5t > t_p, \forall p \in R$. There are at least M contributing scenarios in R. Let T_i^R be the terminals of scenario i in the set of contributors of R at time $5t$. If there is an active scenario moat $\mathcal{M}$ at time t in the ghost process of scenario i such that the terminals of $\mathcal{M}$ are contained in T_i^R, then (Lemma 3.6) the rate at which $\mathcal{M}$ collects cost shares at time t is at most the contribution requested from R to scenario i at time $5t$.

Thus, we can assume that every active scenario moat of scenario i at time t that contains terminals of T_i^R is not contained in R. Among the moats containing terminals of T_i^R, consider the moat $\mathcal{M}'$ that contained a terminal $u \in T_i^R$ and $v \notin T_i^R$ earliest, say at time t_f. There exist a path from u to v using the edges of the merge tree and holes. Suppose the path $\mathcal{P}(u, v) = \langle u = u_0, x_1, y_1, u_1, x_2, y_2, u_2, \ldots, x_s, y_s, u_s = v \rangle$, where $u_i, i = 1, \ldots, s$ is a scenario terminal and x_i, y_i are locations on the boundary of hole $H_i \subset \mathcal{M}'$.

We claim that for all $j = 1, \ldots, s$, either both $x_j, y_j \in R$ or both $x_j, y_j \notin R$. This is because at time t_f both vertices x_j, y_j were part of some tentative facility moat and thus were contained in the same ghost component. Thus, at time $5t > 5t'$ both vertices must be contained in the same real component (due to Lemma 3.6). Thus, there must be an edge (u_{j-1}, x_j) or (y_j, u_j) that crosses R (wlog say (u_{j-1}, x_j)). Thus, we can charge the demand of R at time $5t$ from scenario i to the fractional cost of (u_{j-1}, x_j). The fraction $f(u_{j-1}, x_j)$ corresponding to the edge is greater than the demanding rate of R due to Lemma 3.6.

It is also clear that two different real components R and R' cannot charge to the same portion of an edge of the merge tree of scenario i.

The charge to cost shares for the Steiner forest on the scenario terminals is $8f^1$ (Lemma 3.3); the cost of the portion of the Steiner tree on the open facilities p charged until time t_p is $8f^1$ (Lemma 3.7). Finally, the remaining portion of the tree costs charge to $5 \cdot 4(f^1 + f^2) + 5(f^1 + f^2)$ by Lemma 3.8. This gives the following theorem.

Theorem 3.1. *The above cost-sharing scheme is 2-competitive, cross-monotone and 41-budget balanced.*

4 Cost Shares for CBTAN with Approximate Efficiency

In the previous section, we defined cost-shares for the CBTAN problem that were cross-monotone, and (approximately) budget-balanced. In addition to these two properties, one may also want the cost-shares to give rise to Moulin mechanisms that result in high social welfare.

Definition 4.1. *Suppose each player $i \in U$ has a private utility u_i. For a set $S \subseteq U$, define $u(S) = \sum_{i \in S} u_i$. Define the* social cost *$\Pi(S)$ of a set $S \subseteq U$ as $\Pi(S) = u(U \setminus S) + C(S)$. The Moulin mechanism $M(\xi)$ is said to be* α-approximate *[RS06] if*

$$\Pi(S^M) \leq \alpha \cdot \Pi(S) \quad \forall S \subseteq U.$$

where S^M is the final set of players computed by the Moulin mechanism $M(\xi)$ on U.

We can also prove the following theorem:

Theorem 4.1. *There exist $O(1)$-budget-balanced cross-monotone cost-shares which are also $O(\log^2 k)$-approximate; i.e., their inefficiency is at most $O(\log^2 k)$ times the inefficiency of any cost-sharing mechanism.*

This result extends the recent result of Roughgarden and Sundararajan [RS], who presented a cross-monotone cost-sharing scheme for the Single-Source Rent-or-Buy (SSRoB) problem with an approximate efficiency of $O(\log^2 k)$. We end by noting that while we can define these cost-shares which are cross-monotone and even have a better budget balance factor than the cost shares defined in section 3, we do not yet have an efficient algorithm to compute these cost shares.

References

[AFK+04] Archer, A., Feigenbaum, J., Krishnamurthy, A., Sami, R., Shenker, S.: Approximation and collusion in multicast cost sharing. Games and Economic Behavior 47(1), 36–71 (2004)

[AKR95] Agrawal, A., Klein, P., Ravi, R.: When trees collide: an approximation algorithm for the generalized Steiner problem on networks (Preliminary version in 23rd STOC, 1991). SIAM J. Comput. 24(3), 440–456 (1995)

[AZ02] Andrews, M., Zhang, L.: Approximation algorithms for access network design (Preliminary version in 39th FOCS, 1998). Algorithmica 34(2), 197–215 (2002)

[FKSS03] Feigenbaum, J., Krishnamurthy, A., Sami, R., Shenker, S.: Hardness results for multicast cost-sharing. Theoretical Computer Science 304, 215–236 (2003)

[FPS01] Feigenbaum, J., Papadimitriou, C.H., Shenker, S.: Sharing the cost of multicast transmissions (Special issue on internet algorithms). J. Comput. System Sci. 63(1), 21–41 (2001)

[GKK+01] Gupta, A., Kumar, A., Kleinberg, J., Rastogi, R., Yener, B.: Provisioning a Virtual Private Network: A network design problem for multicommodity flow. In: Proceedings of the 33rd ACM Symposium on the Theory of Computing (STOC), pp. 389–398. ACM Press, New York (2001)

[GKL76] Green, J., Kohlberg, E., Laffont, J.J.: Partial equilibrium approach to the free rider problem. Journal of Public Economics 6, 375–394 (1976)

[GKR03] Gupta, A., Kumar, A., Roughgarden, T.: Simpler and better approximation algorithms for network design. In: Proceedings of the 35th ACM Symposium on the Theory of Computing (STOC), pp. 365–372. ACM Press, New York (2003)

[GPRS04] Gupta, A., Pál, M., Ravi, R., Sinha, A.: Boosted sampling: Approximation algorithms for stochastic optimization problems. In: Proceedings of the 36th ACM Symposium on the Theory of Computing (STOC), pp. 417–426. ACM Press, New York (2004)

[GRS04] Gupta, A., Ravi, R., Sinha, A.: An edge in time saves nine: LP rounding approximation algorithms for stochastic network design. In: Proceedings of the 45th Symposium on the Foundations of Computer Science (FOCS), pp. 218–227 (2004)

[GST04] Gupta, A., Srinivasan, A., Tardos, É.: Cost-sharing mechanisms for network design. In: Jansen, K., Khanna, S., Rolim, J.D.P., Ron, D. (eds.) RANDOM 2004 and APPROX 2004. LNCS, vol. 3122, pp. 139–150. Springer, Heidelberg (2004)

[HST05] Hayrapetyan, A., Swamy, C., Tardos, É.: Network design for information networks. In: Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 933–942. ACM Press, New York (2005)

[IKMM04] Immorlica, N., Karger, D., Minkoff, M., Mirrokni, V.: On the costs and benefits of procrastination: Approximation algorithms for stochastic combinatorial optimization problems. In: Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 684–693. ACM Press, New York (2004)

[IMM05] Immorlica, N., Mahdian, M., Mirrokni, V.S.: Limitations of cross-monotonic cost sharing schemes. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 602–611. ACM Press, New York (2005)

[JV01] Jain, K., Vazirani, V.: Applications of approximation algorithms to cooperative games. In: Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing (STOC), pp. 364–372. ACM Press, New York (2001)

[KLS05] Könemann, J., Leonardi, S., Schäfer, G.: A group-strategyproof mechanism for Steiner forests. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 612–619. ACM Press, New York (2005)

[KLSvZ05] Könemann, J., Leonardi, S., Schäfer, G., van Zwam, S.: From primal-dual to cost shares and back: a stronger LP relaxation for the Steiner forest problem. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 930–942. Springer, Heidelberg (2005)

[KM00] Karger, D.R., Minkoff, M.: Building Steiner trees with incomplete global knowledge. In: Proceedings of the 41th Symposium on the Foundations of Computer Science (FOCS), pp. 613–623 (2000)

[KSK96] Kent, K.J., Skorin-Kapov, D.: Population monotonic cost allocations on MSTs. In: Proceedings of the 6th International Conference on Operational Research (Rovinj, 1996). Croatian Oper. Res. Soc., Zagreb, pp. 43–48 (1996)

[LS04] Leonardi, S., Schäfer, G.: Cross-monotonic cost sharing methods for connected facility location games. Theor. Comput. Sci. 326(1-3), 431–442 (2004)

[Mou99] Moulin, H.: Incremental cost sharing: Characterization by coalition strategyproofness. Social Choice and Welfare 16, 279–320 (1999)

[MS01] Moulin, H., Shenker, S.: Strategyproof sharing of submodular costs: budget balance versus efficiency. Econom. Theory 18(3), 511–533 (2001)

[PT03] Pál, M., Tardos, É.: Group strategyproof mechanisms via primal-dual algorithms. In: Proceedings of the 44th Symposium on the Foundations of Computer Science (FOCS), pp. 584–593 (2003)

[Rob79] Roberts, K.: The characterization of implementable choice rules. In: Laffont, J.J. (ed.) Aggregation and Revelation of Preferences, North-Holland, Amsterdam (1979)

[RS] Roughgarden, T., Sundararajan, M.: Approximately efficient cost-sharing mechanisms. (Unpublished manuscript)

[RS99] Ravi, R., Salman, F.S.: Approximation algorithms for the traveling purchaser problem and its variants in network design. In: Nešetřil, J. (ed.) ESA 1999. LNCS, vol. 1643, pp. 29–40. Springer, Heidelberg (1999)

[RS06] Roughgarden, T., Sundararajan, M.: New trade-offs in cost-sharing mechanisms. In: STOC (2006)

[SK04] Swamy, C., Kumar, A.: Primal-dual algorithms for connected facility location problems. Algorithmica 40(4), 245–269 (2004)

Distance Coloring

Alexa Sharp

Cornell University, Ithaca, NY 14853
asharp@cs.cornell.edu

Abstract. Given a graph $G = (V, E)$, a (d, k)-coloring is a function from the vertices V to colors $\{1, 2, \ldots, k\}$ such that any two vertices within distance d of each other are assigned different colors. We determine the complexity of the (d, k)-coloring problem for all d and k, and enumerate some interesting properties of (d, k)-colorable graphs. Our main result is the discovery of a dichotomy between polynomial and NP-hard instances: for fixed $d \geq 2$, the distance coloring problem is polynomial time for $k \leq \lfloor \frac{3d}{2} \rfloor$ and NP-hard for $k > \lfloor \frac{3d}{2} \rfloor$.

Keywords: Graph coloring, power graphs, complexity threshold.

1 Introduction

The classic *k-coloring* problem tries to assign a color from 1 to k to each vertex in a graph such that no two adjacent vertices share the same color [1]. The k-coloring problem, along with many variations and generalizations, is well-studied in both computer science and mathematics [2,3,4,5,6]. Its applications range from frequency assignment [7] to circuit board testing [8], among others.

The *distance* (d, k)*-coloring* problem is a generalization of k-coloring that tries to assign a color from 1 to k to each vertex such that no two vertices within distance d of each other share the same color. Clearly, k-coloring is a special case of (d, k)-coloring with $d = 1$. Conversely, (d, k)-coloring a graph G is equivalent to k-coloring G^d, the dth power graph of G. (The graph G^d has the same vertex set as G and an edge between two vertices if and only if they are within distance d of each other in G.) In this way (d, k)-coloring is also a special case of k-coloring.

Although k-coloring is NP-complete for $k \geq 3$ and polynomial-time for $k \leq 2$ [1], the complexity of (d, k)-coloring is not so straightforward. It is NP-complete for large k [9,10]; other special cases have been studied in depth, such as cubic graphs [11], planar graphs [12], and trees [13,14]. However, there are many values of d and k for which (d, k)-coloring is polynomial-time; the dichotomy between polynomial and NP-hard instances is the subject of this paper. The results are not only of theoretical interest, but also of practical use in applications with underlying structures that fit the power graph model. For example, we may want to assign k frequencies to a set of radio stations, but require that any two stations within distance d of each other use different frequencies.

The main result is significant, as it classifies the complexity of all instances of (d, k)-coloring for $d \geq 2$: determining whether a graph is (d, k)-colorable is

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 510–521, 2007.

polynomial-time for $k \leq \lfloor \frac{3d}{2} \rfloor$, but NP-hard for $k > \lfloor \frac{3d}{2} \rfloor$. Moreover, (d,k)-coloring on trees is solvable in polynomial time.

This paper presents many opportunities for future work; further improvements on these results, as well as their implications to generalized graph decompositions and complexity are explored in [15].

2 Preliminaries

Given a graph $G = (V, E)$ and integers $d \geq 1$ and $k \geq d+1$, the goal of (d,k) *coloring* is to find an assignment of k colors to the vertices of G such that no two vertices within distance d of each other share the same color. More precisely, we want a function $f : V \to \{1, 2, \ldots, k\}$ such that $d(u,v) \leq d \Rightarrow f(u) \neq f(v)$. We say G is (d,k)-*colorable* if such a function exists, and call the function itself a (d,k)-*coloring*. The standard k-coloring problem is the special case when $d = 1$.

Observe that each connected component of G can be colored independently; moreover, any component of size less than or equal to k can be (d,k)-colored by assigning a distinct color to each vertex. Thus in this paper we assume that G is connected and has at least $k+1$ vertices.

Definition 1. *For a vertex v and integer r, let $G_v^r \subseteq G$ be the* subgraph of radius r around v*, that is, the subgraph induced by $\{w \mid d(w,v) \leq r\}$.*

Definition 2. *For a subgraph $G' \subseteq G$, the* diameter of G'*, denoted $diam(G')$, is the maximum shortest path distance between any two vertices of G', that is, $diam(G') = \max\{d(u,v) \mid u, v \in G'\}$.*

Definition 3. *Given a connected graph $G = (V, E)$, the set $V' \subseteq V$ is a* cutset *if the subgraph induced by $V \setminus V'$ is no longer connected.*

Definition 4. *Given a graph G of size at least $k+1$, a graph $G' = (V', E') \subseteq G$ is a* forbidden (d,k) subgraph *if $diam(G') \leq \min\{d, |V'| - (k-d) - 1\}$.*

2.1 Properties of (d,k)-Colorable Graphs

Theorem 1. *If G contains a forbidden (d,k) subgraph then it is not (d,k)-colorable.*

Proof. Suppose $G' = (V', E')$ is a subgraph of G with $diam(G') \leq \min\{d, |V'| - (k-d) - 1\}$. Let G'' be a connected graph induced by V' and $\max\{0, k+1-|V'|\}$ vertices of $V \setminus V'$; G'' has $\geq k+1$ vertices and diameter at most d, which precludes a (d,k)-coloring. □

Corollary 1. *If there is a vertex $v \in G$ such that $|G_v^r| \geq (k-d) + 2r + 1$ for any $1 \leq r \leq \lfloor \frac{d}{2} \rfloor$ then G cannot be (d,k)-colored.*

Proof. The diameter of G_v^r is at most $2r \leq d$. But $2r = (k - d + 2r + 1) - (k - d) - 1 \leq |G_v^r| - (k-d) - 1$ and Theorem 1 completes the proof. □

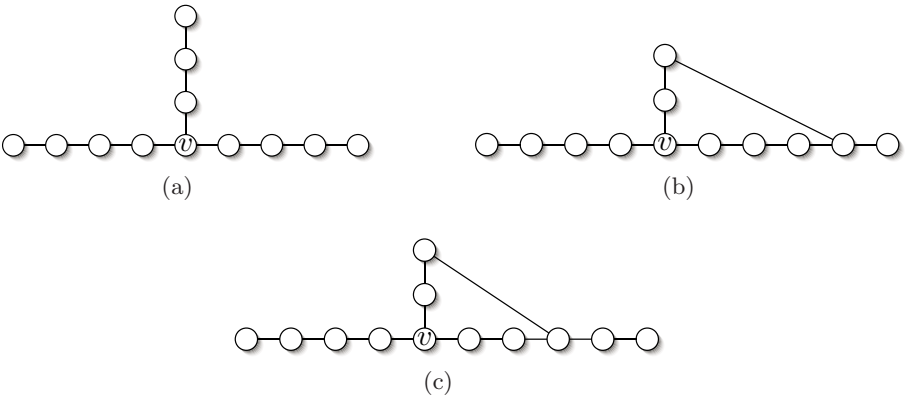


Fig. 1. Examples of Corollary 1 for $(6,9)$-coloring. The graphs of 1(a), 1(b) cannot be $(6,9)$-colored because $|G_v^3| = 10$; the graph of 1(c) is $(6,9)$-colorable because $|G_v^r| \leq 2r+3$ for all $r \leq 3$.

Theorem 2. *Given a (d,k)-colorable graph G with $k \leq \lfloor \frac{3d}{2} \rfloor$, if G_v^{k-d} is a strict subgraph of G for some $v \in G$, then G_v^{k-d} contains a cutset of size ≤ 2 disconnecting v from $G \setminus G_v^{k-d}$.*

Proof. First observe that any vertex $y \in G \setminus G_v^{k-d}$ is connected to v through at least one vertex of each $G_v^i \setminus G_v^{i-1}$ for $1 \leq i \leq k-d$. We will show there is $1 \leq i \leq k-d$ such that $|G_v^i \setminus G_v^{i-1}| \leq 2$. Then the removal of $G_v^i \setminus G_v^{i-1}$, of at most two vertices, would disconnect y from v.

By way of contradiction, suppose $|G_v^i \setminus G_v^{i-1}| \geq 3$ for all $1 \leq i \leq k-d$. Then $|G_v^0| = 1$ implies $|G_v^i| \geq 3i+1$. Hence $|G_v^{k-d}| \geq 2(k-d)+(k-d)+1$, and G is not (d,k)-colorable by Corollary 1, a contradiction. □

Lemma 1. *Let P be a path of length $0 \leq p \leq \lfloor \frac{d}{2} \rfloor$, with left and right endpoints v_L and v_R, respectively, in a (d,k)-colorable graph G, $k \leq \lfloor \frac{3d}{2} \rfloor$. Suppose there exist disjoint subgraphs P_L, P_R in $G \setminus P$ such that $P_L \cup \{v_L\}$, $P_R \cup \{v_R\}$ are connected and $|P_L|, |P_R| \geq \lfloor \frac{d}{2} \rfloor$; let O be the non-P vertices connected to P in the graph $G \setminus (P_L \cup P_R)$. Then $|O| \leq k-d-1$.*

Proof. See Appendix for the proof and an illustration of the notation. □

3 Negative Complexity Results

We know that $(1,k)$-coloring is NP-hard for $k \geq 3$; we now use the results of Section 2 to show that for $d \geq 2$, (d,k)-coloring is NP-hard for $k > \lfloor \frac{3d}{2} \rfloor$.

Theorem 3. *The (d,k)-coloring problem is NP-hard for $d \geq 2$, $k > \lfloor \frac{3d}{2} \rfloor$.*

Theorem 3 follows via a reduction from $(1,k)$-coloring (k-COL). Given a graph G that we wish to k-color, we construct a graph G' such that G is k-colorable if and only if G' is (d,k)-colorable. The building block of this reduction is a *triangle gadget* G^{Δ}, which is shown in Figure 2.

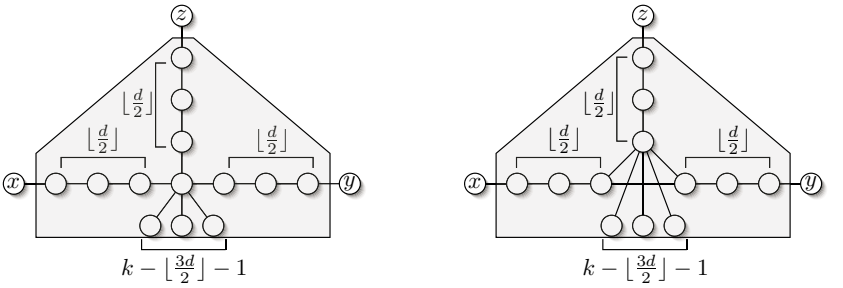


Fig. 2. 2(a) and 2(b) show Theorem 3's G^{Δ} gadget for odd and even d, respectively

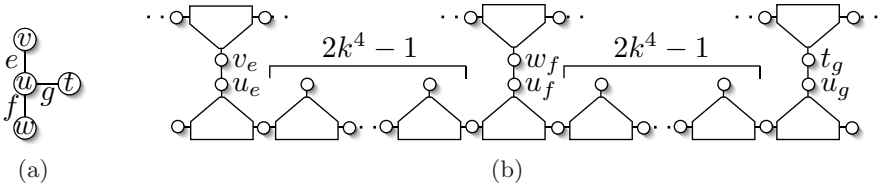


Fig. 3. 3(a) An instance G of k-COL. 3(b) A subgraph of the graph G' constructed from G in Theorem 3.

Lemma 2. *If G^{Δ} is (d,k)-colorable then x, y and z have the same color.*

Proof. The $k-1$ vertices in $G^{\Delta} \setminus \{x, y, z\}$ are within distance d of each other, and use $k-1$ distinct colors. The vertices x, y and z are within distance d of all these $k-1$ colors, but distance $d+1$ from each other. If G^{Δ} is (d,k)-colorable then x, y and z are colored the same. □

Reduction from k-COL. Given an instance $G = (V, E)$ of k-COL, create the graph $G' = (V', E')$ as follows. For each vertex $u \in V$, create gadget $G^u \subseteq G'$ by concatenating $deg(u) \cdot 2k^4$ copies of G^{Δ}, overlapping the x and y vertices, always leaving the z vertices open. Every $2k^4 th$ z vertex is reserved for use as follows: for each edge $e = (u, v) \in G$, create an edge $(u_e, v_e) \in G'$ where u_e and v_e are reserved z vertices of G^u and G^v, respectively; an example of this reduction is shown in Figure 3. Note that G' is polysize, as $|G^{\Delta}| = k + 2$ and it is copied $\sum_{u \in V} deg(u) \cdot 2k^4$ times.

Lemma 3. *If G' is (d,k)-colorable then $u_e, u_f \in G'$ have the same color for all edges e, f incident to $u \in G$.*

Proof. By Lemma 2 we know that G^u's x, y and z vertices must be the same color. The vertices u_e and u_f are simply z vertices, and the result follows. □

Lemma 4. *If G' is (d,k)-colorable then for edge $e = (u, v) \in G$, $u_e, v_e \in G'$ are different colors.*

Proof. If $e = (u, v) \in G$ then $(u_e, v_e) \in G'$. Thus $d(u_e, v_e) \leq d$ and u_e, v_e are different colors in G'. □

Lemma 5. *If G' is (d,k)-colorable then G is k-colorable.*

Proof. If G' has a (d,k)-coloring C then create a k-coloring D of G by setting $D(u) = C(u_e)$ for any e incident to u. Since $C(u_e) = C(u_f)$ for all e, f incident to u by Lemma 3, D is well-defined. Moreover, $D(u) \neq D(v)$ for $(u,v) \in G$ by Lemma 4, and D uses at most k colors. □

Lemma 6. *If G is k-colorable then G' is (d,k)-colorable.*

The proof of Lemma 6 requires some additional framework. Consider the colors of the vertices of $G^\Delta \setminus \{x,y,z\}$, labeled as in figure 4(a). If we use the coloring of this G^Δ to color an adjacent G^Δ as shown in Figures 4(b) or 4(c) then we will have defined a symmetry group based on the colors of $G^\Delta \setminus \{x,y,z\}$; the base set is $A = \{a_1, a_2, \ldots, a_{k-1}\}$ and we have two elements $\sigma, \pi : A \to A$, where σ is the shift operator $(a_1\ a_2\ \cdots\ a_{k-1})$ and τ is the adjacent transposition operator $(a_1\ a_2)(a_3)(a_4)\cdots(a_{k-1})$.

Lemma 7. *Applying either the shift σ or the transposition τ to the colors of G^Δ yields a valid coloring.*

Proof. The vertices of the same color in adjacent G^Δs are at least distance $d+1$ apart. □

Lemma 8. *Any adjacent transposition $\pi = (a_j\ a_{j+1})$ can be generated from k compositions of σ and τ.*

Proof. Use the shift σ $j-1$ times, transpose with τ once, then shift again $(k-1)-(j-1)$ times. The only elements transposed are j and $j+1$; the remaining elements shift $k-1$ times and hence are unchanged. □

Lemma 9. *Any transposition $\pi = (a_i\ a_j)$ can be generated from $\leq 2k$ compositions of adjacent transpositions.*

Proof. Without loss of generality, suppose $i < j$. Then

$$(a_i a_j) = (a_{j-1}a_j)(a_{j-2}a_{j-1})\cdots(a_i a_{i+1})(a_{i+1}a_{i+2})\cdots(a_{j-2}a_{j-1})(a_{j-1}a_j),$$

which uses $2(j-i)-1 \leq 2k$ transpositions, as required. □

Lemma 10. *Any cycle $\pi = (\pi_1\ \pi_2\ \cdots\ \pi_p)$ can be generated from $\leq p$ compositions of transpositions.*

Proof. Note that $(\pi_1\ \pi_2\ \cdots\ \pi_p) = (\pi_1\ \pi_p)(\pi_1\ \pi_{p-1})\cdots(\pi_1\ \pi_3)(\pi_1\ \pi_2)$. □

Lemma 11. *Any permutation on k elements can be generated from $\leq 2k^4$ compositions of the shift σ and the adjacent transposition τ.*

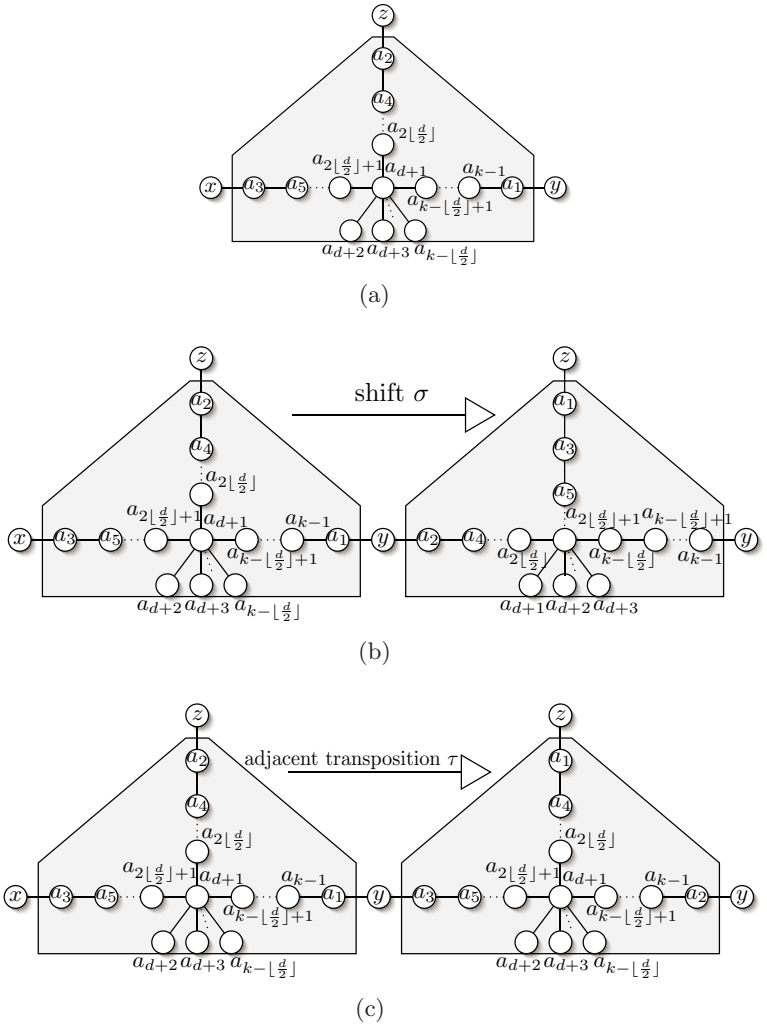


Fig. 4. 4(a) The set A for the basis of the permutation group for odd d. The same labeling can be used for even d, ignoring the center vertex. 4(b) The shift operator σ. 4(c) The adjacent transposition operator τ.

Proof. First note that any permutation on A can be generated by at most k cycles of length k. By Lemmas 8-10 we have that

$$\begin{aligned} k \text{ cycles of length } k \text{ are generated by } &\leq k^2 \text{ transpositions,} \\ \text{which are generated by } &\leq 2k^3 \text{ adjacent transpositions,} \\ \text{which are generated by } &\leq 2k^4 \text{ compositions of } \sigma \text{ and } \tau. \end{aligned}$$

Thus any permutation is generated by $\leq 2k^4$ compositions of σ and τ. □

Proof of Lemma 6. Given a k-coloring D of G we show how to (d,k)-color the gadgets of G'. First, for each vertex $u \in G$, color all of G^u's x, y and z vertices with the color $D(u)$. This is feasible because these vertices are distance $d+1$ apart, and their color is different from all neighboring gadgets' x, y, z colors. Next, for each vertex $u \in G$, color the G^Δ's directly connected to G^v for some $v \neq u \in G$. Now the remaining uncolored G^Δ gadgets in G^u are chains of $2k^4-1$ G^Δs between two colored G^Δs. We know from Lemma 11 that there is some chain of at most $2k^4$ compositions of σ and τ that lead from any one coloring of G^Δ to the next, and each composition corresponds to a valid coloring of each G^Δ. Thus we know there is a sequence of colorings getting us from one G^Δ to the one distance $2k^4$ away, and G' can thus be (d,k) colored. □

Proof of Theorem 3. The polynomial reduction constructs a graph G' that is (d,k)-colorable if and only if G is k-colorable, by Lemmas 5 and 6. Therefore (d,k)-coloring is NP-complete for $d \geq 2$ and $k \geq \lfloor \frac{3d}{2} \rfloor + 1$. □

4 Positive Complexity Results

Thus (d,k)-coloring is NP-hard for $d \geq 2, k > \lfloor \frac{3d}{2} \rfloor$; we now show that the remaining parameters lead to polynomial-time solutions.

4.1 $(d, d+1)$-Coloring

Theorem 4. *The $(d, d+1)$-coloring problem is polynomial-time solvable.*

Proof. A graph is $(1,2)$-colorable if and only if it is bipartite. For $d \geq 2$, if G contains a vertex of degree 3 or greater, then it is not $(d, d+1)$-colorable by Corollary 1. Otherwise, G is a path or cycle. If it is a cycle but its length is not a multiple of $(d+1)$ then it is not $(d, d+1)$-colorable. Otherwise G is a path or a cycle whose length is a multiple of $(d+1)$. In either case, cycle through the colors $1, 2, \ldots, (d+1)$, which ensures that vertices within distance d of each other are colored differently. □

4.2 (d,k)-Coloring for $k \leq \lfloor \frac{3d}{2} \rfloor$

We describe an algorithm polynomial in $|G|$ that either produces a (d,k)-coloring of G or declares that no such coloring exists. The algorithm finds a bounded tree-width tree decomposition of G [16], then colors the graph using known coloring algorithms for bounded tree-width graphs [17,18].

Definition 5. *A* tree decomposition *of $G = (V, E)$ is a triple (T, F, X) consisting of an undirected tree (T, F) and a map $X : T \to 2^V$ associating a subset $X_i \subseteq V$ with each $i \in T$ such that*

I. $V = \bigcup_{i \in T} X_i$;
II. *for all edges $(u, v) \in E$, there exists $i \in T$ such that $u, v \in X_i$;*
III. *if j lies on the path between i and k in (T, F), then $X_i \cap X_k \subseteq X_j$.*

Definition 6. *The* width *of* (T, F, X) *is* $\max_i\{|X_i| - 1\}$.

Definition 7. *A* path decomposition (T, F, X) *is a tree decomposition in which the graph* (T, F) *is a simple path.*

We first try to compute a path decomposition of the graph G. Let P be a simple path of length $\geq d + 1$ in G, and let s be a center vertex; if no such path exists then $diam(G) \leq d$ and G is not (d, k)-colorable. Otherwise, perform a breadth-first search on G starting from s to get *level sets* $L_0, L_1, \ldots, L_m$, where L_i consists of the set of vertices of distance i from the root s. The *level graph* H is the graph consisting of vertices V and *directed edges* from L_i to L_{i+1}, ignoring edges between vertices on the same level. We take $L_j = \emptyset$ for $j > m$. For $0 \leq i \leq \max\{0, m - d\}$, let

$$X_i \stackrel{def}{=} \bigcup_{j=i}^{i+d} L_j \ . \tag{1}$$

Lemma 12. *If* G *is* (d, k)*-colorable, then* $|X_i| \leq 5d$.

Proof. Label the right side of P as $s = s_0^R, s_1^R, \ldots, s_{m_R}^R$ and the left side of P as $s = s_0^L, s_1^L, \ldots, s_{m_L}^L$ such that $s_i^R, s_i^L \in L_i$ (where $m_L, m_R \geq \lceil \frac{d}{2} \rceil$ because $\ell(P) \geq d + 1$). For $0 \leq i \leq m$, let T_i^R, T_i^L be the subgraphs of the level graph rooted at s_i^R, s_i^L, respectively, consisting of all vertices reachable from s_i^R, s_i^L by a (directed) path in H. Let $T^L = H \setminus T_1^R$, $T^R = H \setminus T_1^L$. Note that $H = T^L \cup T^R$. See Figure 5.

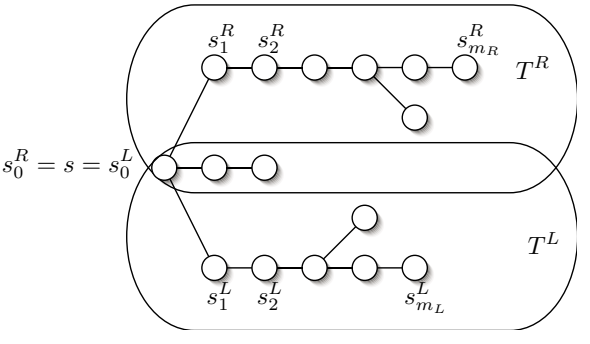


Fig. 5. Notation used in Lemma 12

To obtain the result for $|X_i|$, we bound $|X_i \cap T^R|$ and $|X_i \cap T^L|$ separately, showing $|X_i \cap T^R|, |X_i \cap T^L| \leq \lfloor \frac{5d}{2} \rfloor$. The arguments are identical except for notation, so without loss of generality we argue only the former.

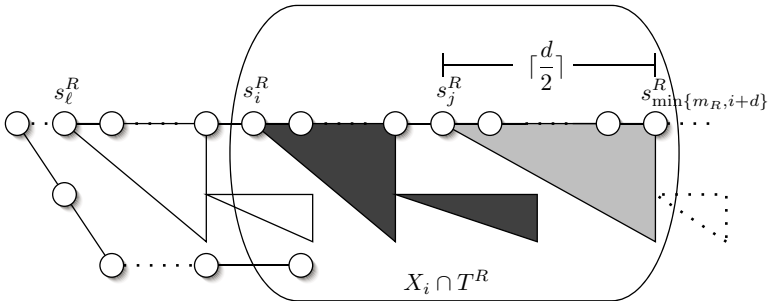


Fig. 6. The above inequalities shown pictorially. The light gray triangle is $X_i \cap T_j^R$. The dark gray triangles are $X_i \cap (T_i^R \setminus T_j^R)$. The small white triangle is $X_i \cap (T_\ell^R \setminus T_i^R)$.

For $|X_i \cap T^R|$, let $j = \min\{m_R, i+d\} - \lceil \frac{d}{2} \rceil$ and $\ell = \max\{0, i-(k-d)\}$ (see Figure 6). Then $j \geq 0$ and $\ell \leq i \leq m_R - \lceil \frac{d}{2} \rceil$, and

$$|X_i \cap T^R| = |X_i \cap T_i^R| + |X_i \cap (T^R \setminus T_i^R)| \tag{2}$$
$$= |X_i \cap T_j^R| + |X_i \cap (T_i^R \setminus T_j^R)| + |X_i \cap (T^R \setminus T_i^R)| \tag{3}$$
$$\leq |X_i \cap T_j^R| + |X_i \cap (T_i^R \setminus T_j^R)| + |X_i \cap (T_\ell^R \setminus T_i^R)|, \tag{4}$$

where the last inequality follows because any vertex $v \in (X_i \cap T^R) \setminus T_\ell^R$ is at least distance $i - \ell + 1 \geq (k-d)$ from its connection point on P, which has $\lfloor \frac{d}{2} \rfloor$ and $\lceil \frac{d}{2} \rceil$ vertices of P on either side of it; this forms a forbidden subgraph, therefore no such v exists since G is (d,k)-colorable.

For $|X_i \cap T_j^R|$, note that there are $\lfloor \frac{d}{2} \rfloor$ vertices of $P \setminus T_j^R$ within $\lfloor \frac{d}{2} \rfloor$ of s_j^R. Either G is not (d,k)-colorable or there are $\leq k - \lfloor \frac{d}{2} \rfloor \leq d$ vertices in T_j^R within distance $\lceil \frac{d}{2} \rceil$ of s_j^R; since $T_j^R \cap X_i$ is precisely this set of vertices, $|T_j^R \cap X_i| \leq d$.

For $|X_i \cap (T_i^R \setminus T_j^R)|$, consider the interval of length $p = j - i \leq \lfloor \frac{d}{2} \rfloor$ from s_i^R to s_j^R. There are at least $\lfloor \frac{d}{2} \rfloor$ vertices in $P \setminus T_i^R$ and $P \cap T_j^R$, respectively, thus by Lemma 1, either G is not (d,k)-colorable or $|(T_i^R \setminus T_j^R) \setminus P| \leq k - d - 1$, and $|X_i \cap (T_i^R \setminus T_j^R)| \leq k - d - 1 + p + 1 \leq k - \lceil \frac{d}{2} \rceil \leq d$.

For $|X_i \cap (T_\ell^R \setminus T_i^R)|$ consider the interval s_ℓ^R to s_i^R of length $p = i - \ell \leq \lfloor \frac{d}{2} \rfloor$. There are $\geq \lfloor \frac{d}{2} \rfloor$ vertices in $P \setminus T_\ell^R$ and $P \cap T_i^R$, respectively, and either G is not (d,k)-colorable, or $|T_\ell^R \setminus T_i^R \setminus P| \leq k - d - 1$, and $|X_i \cap (T_\ell^R \setminus T_i^R)| \leq \lfloor \frac{d}{2} \rfloor$.

Thus $|X_i \cap T^R| \leq \lfloor \frac{5d}{2} \rfloor$, and $|X_i \cap H| = |X_i| \leq 5d$, as required. □

Theorem 5. *For fixed constants d, k with $k \leq \lfloor \frac{3d}{2} \rfloor$, there is an $O(n)$ algorithm for finding a (d,k)-coloring of G if one exists.*

Proof. We use the algorithm of Section 4.2 to either find a path decomposition (T, F, X) of G of width at most $5d$ or determine that G is not (d,k)-colorable. Assume the former. Then (T, F, X) is also a path decomposition of the dth power graph G^d. The algorithms of [17,18] can determine the $(1,k)$-colorability of G^d in linear time, and a $(1,k)$-coloring of G^d is precisely a (d,k)-coloring of G. □

5 Coloring on Trees

Although it is already known that (d, k)-coloring trees is polynomial-time solvable [13], the results of this paper lead to a nice combinatorial method to achieve the same result. This does not require d or k to be fixed.

Algorithm to Find a (d, k)-coloring for a Tree T.

1. If $d = 1$ and $k \geq 2$ then we $(1, 2)$-color T, a bipartite graph.
2. Otherwise, perform a breadth-first search on T, starting from a leaf node s to get level sets $L_0, L_1, \ldots, L_m$, where L_i consists of the set of vertices of distance i from the root s.
3. For $i = 0, 1, 2, \ldots, m$, color the vertices of L_i in some arbitrary order by assigning $v \in L_i$ the lowest color not yet used in G_v^d.
4. If we need more than k colors then T is not (d, k)-colorable, otherwise return the assigned coloring.

Lemma 13. *The algorithm returns a coloring of T if and only if T is (d, k)-colorable.*

Proof. [⇒] Consider a coloring found by the algorithm; it uses at most k colors. Moreover, no two vertices within distance d of each other are assigned the same color. Indeed, consider any two vertices u and v such that $d(u, v) \leq d$, and suppose u preceded v in the algorithm. Then when v is considered, u's color is excluded from consideration; consequently, the algorithm will not assign v the color that it used for u.

[⇐] Suppose T is (d, k)-colorable. Consider one of the vertices v, and suppose there are t vertices already colored within distance d of v. Then these t vertices along with v are within distance d of each other by properties of the BFS tree, and hence $t+1 \leq k$ necessarily. It follows that this set of t vertices uses $\leq t \leq k-1$ colors, and so there is a color that can be assigned to v without exceeding the allotted k colors. □

This work was supported by NSF grant CCF-0635028. Any views and conclusions expressed herein are those of the author and do not necessarily represent the official policies or endorsements of the National Science Foundation or the United States government.

Acknowledgments. I would like to thank Dexter Kozen for his many useful comments and insights, as well as the anonymous reviewers for their helpful suggestions.

References

1. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1990)
2. Bondy, J., Murty, U.: Graph Theory with Applications. MacMillan Press Ltd, NYC (1978)

3. Fiorini, S., Wilson, R.J: Edge-colourings of graphs. In: Beineke, L.W., Wilson, R.J. (eds.) Selected Topics in Graph Theory, pp. 103–126. Academic Press, Inc, London (1978)
4. Bollobás, B., Harris, A.J.: List-colourings of graphs. Graphs and Combinatorics 1, 115–127 (1985)
5. Wilson, B.: Line-distinguishing and harmonious colourings. In: Nelson, R., Wilson, R.J. (eds.) Longman Scientific & Technical, Longman house, Burnt Mill, Harlow, Essex, UK. Graph Colourings. Pitman Research Notes in Mathematics Series, pp. 115–133 (1990)
6. Chetwynd, A.: Total colourings of graphs. In: Nelson, R., Wilson, R.J., (eds.)Graph Colourings. Pitman Research Notes in Mathematics Series. Longman Scientific & Technical, Longman house, Burnt Mill, Harlow, Essex, UK, pp. 65–77 (1990)
7. Gamst, A.: Some lower bounds for a class of frequency assignment problems. IEEE Trans. Veh. Technol. VT-35, 8–14 (1986)
8. Garey, M.R., Johnson, D.S., So, H.C: An application of graph coloring to printed circuit testing. IEEE Transactions on Circuits and Systems CAS-23, 591–598 (1976)
9. Lin, Y.-L., Skiena, S.S.: Algorithms for square roots of graphs. SIAM J. Discret. Math. 8, 99–118 (1995)
10. McCormic, S.T.: Optimal approximation of sparse hessians and its equivalence to a graph coloring problem. Math. Programming 26, 153–171 (1983)
11. Heggernes, P., Telle, J.A.: Partitioning graphs into generalized dominating sets. Nordic J. of Computing 5, 128–142 (1998)
12. Agnarsson, G., Halldórsson, M.M.: Coloring powers of planar graphs. In: Proceedings of the 11th annual ACM-SIAM symposium on Discrete algorithms, pp. 654–662. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2000)
13. Bertossi, A.A., Pinotti, M.C., Rizzi, R.: Channel assignment on strongly-simplicial graphs. In: IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, p. 222. IEEE Computer Society Press, Los Alamitos (2003)
14. Agnarsson, G., Greenlow, R., Halldórsson, M.: On powers of chordal graphs and their colorings. Congressus Numerantium 100, 41–65 (2000)
15. Kozen, D., Sharp, A.: On distance coloring. Technical Report cul.cis/TR2007-2084, Cornell University (2007)
16. Robertson, N., Seymour, P.D.: Graph minors. ii. algorithmic aspects of tree-width. J. Algorithms 7, 309–322 (1986)
17. Andrzejak, A.: An algorithm for the tutte polynomials of graphs of bounded treewidth. Discrete Math. 190, 39–54 (1998)
18. Noble, S.D.: Evaluating the tutte polynomial for graphs of bounded tree-width. Comb. Probab. Comput. 7, 307–321 (1998)

Appendix

Lemma 14. *Let P be a path of length $0 \leq p \leq \lfloor \frac{d}{2} \rfloor$, with left and right endpoints v_L and v_R, respectively, in a (d,k)-colorable graph G, $k \leq \lfloor \frac{3d}{2} \rfloor$. Suppose there exist disjoint subgraphs P_L, P_R in $G \setminus P$ such that $P_L \cup \{v_L\}$, $P_R \cup \{v_R\}$ are connected and $|P_L|, |P_R| \geq \lfloor \frac{d}{2} \rfloor$; let O be the non-P vertices connected to P in the graph $G \setminus (P_L \cup P_R)$. Then $|O| \leq k - d - 1$.*

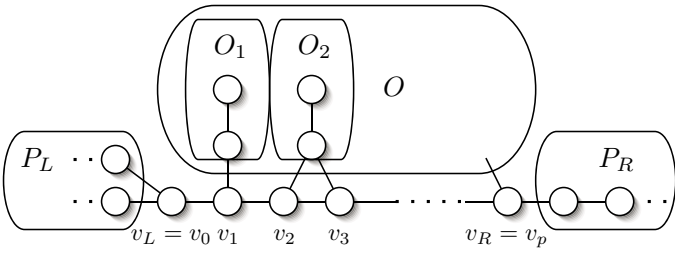


Fig. 7. Notation for Lemma 14

Proof. Label P as $v_L = v_0, v_1, \ldots, v_p = v_R$. Let $O_0, O_1, \ldots, O_p$ be a partition of O such that the induced subgraph $O_i \cup \{v_i\}$ is connected for all $0 \leq i \leq p$. See Figure 7.

Note that $|O_i| \leq k - d - 1 \leq \lfloor \frac{d}{2} \rfloor - 1$ for all $i \geq 0$ otherwise $G_{v_i}^{\lfloor \frac{d}{2} \rfloor}$ would be a forbidden subgraph and G not (d,k)-colorable by Theorem 1. Let

$$d_L = \max\{2\lfloor \frac{d}{2} \rfloor - \max_{i \geq 0}\{|O_i| + i, \lfloor \frac{d}{2} \rfloor\}, \max_{i \geq 0}\{|O_i| - i\}\} \quad \geq \quad \lfloor \frac{d}{2} \rfloor - p \; . \tag{5}$$

Consider the subgraph G' consisting of P, the $d_L \leq \lfloor \frac{d}{2} \rfloor$ closest vertices of P_L to v_L and the $2\lfloor \frac{d}{2} \rfloor - p - d_L \leq \lfloor \frac{d}{2} \rfloor$ closest vertices of P_R to v_R; G' consists of $2\lfloor \frac{d}{2} \rfloor + 1$ vertices within distance $2\lfloor \frac{d}{2} \rfloor$ of each other. The vertices of O are within $d_L + \max_{j \geq 0}\{|O_j| + j\}$ of the $G' \cap P_L$ vertices, where

$$d_L + \max_{j \geq 0}\{|O_j| + j\} = \max\{2\lfloor \frac{d}{2} \rfloor - \max_{i \geq 0}\{|O_i| + i, \lfloor \frac{d}{2} \rfloor\} + \max_{j \geq 0}\{|O_j| + j\}, \tag{6}$$

$$\max_{i \geq 0}\{|O_i| - i\} + \max_{j \geq 0}\{|O_j| + j\}\} \tag{7}$$

$$\leq \max\{2\lfloor \frac{d}{2} \rfloor, \max_{i \geq 0, j \neq i}\{2|O_i|, |O_i| + |O_j| + p\}\} \tag{8}$$

$$\leq \max\{2\lfloor \frac{d}{2} \rfloor, |O| + p\} \; . \tag{9}$$

Similarly, the vertices of O are within $2\lfloor \frac{d}{2} \rfloor - p - d_L + \max_{j \geq 0}\{|O_j| + p - j\}$ of the $G' \cap P_R$ vertices, where

$$2\lfloor \frac{d}{2} \rfloor - d_L + \max_{j \geq 0}\{|O_j| - j\} = 2\lfloor \frac{d}{2} \rfloor - \max\{2\lfloor \frac{d}{2} \rfloor - \max_{i \geq 0}\{|O_i| + i, \lfloor \frac{d}{2} \rfloor\} \tag{10}$$

$$- \max_{j \geq 0}\{|O_j| - j\}, 0\} \tag{11}$$

$$\leq 2\lfloor \frac{d}{2} \rfloor \; . \tag{12}$$

Lastly, the vertices of O are at most distance $p + |O|$ from each other. Thus we have $2\lfloor \frac{d}{2} \rfloor + 1 + |O|$ vertices within distance $\leq \max\{2\lfloor \frac{d}{2} \rfloor, |O| + p\}$ of each other, and hence $|O| \leq k - d - 1$ otherwise there is a forbidden subgraph. □

An $O(\log^2 k)$-Competitive Algorithm for Metric Bipartite Matching

Nikhil Bansal[1], Niv Buchbinder[2], Anupam Gupta[3], and Joseph (Seffi) Naor[4]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
[2] Computer Science Department, Technion, Haifa, Israel
[3] Department of Computer Science Carnegie Mellon University
[4] Microsoft Research, Redmond, WA. On leave from the CS Dept., Technion, Haifa, Israel

Abstract. We consider the online metric matching problem. In this problem, we are given a graph with edge weights satisfying the triangle inequality, and k vertices that are designated as the right side of the matching. Over time up to k requests arrive at an arbitrary subset of vertices in the graph and each vertex must be matched to a right side vertex immediately upon arrival. A vertex cannot be rematched to another vertex once it is matched. The goal is to minimize the total weight of the matching.

We give a $O(\log^2 k)$ competitive randomized algorithm for the problem. This improves upon the best known guarantee of $O(\log^3 k)$ due to Meyerson, Nanavati and Poplawski [19]. It is well known that no deterministic algorithm can have a competitive less than $2k - 1$, and that no randomized algorithm can have a competitive ratio of less than $\ln k$.

1 Introduction

Matching is one of the most fundamental and well-studied optimization problems and it has played a major role in the development of the theory of algorithms; see, e.g., [22] for the history as well as many details and algorithms. In this paper, we consider an *online version of the matching problem* which was first introduced by Khuller, Mitchell and Vazirani [14], and independently by Kalyanasundaram and Pruhs [9]. In this version, the input consists of an edge-weighted graph with k vertices designated as *right-hand side vertices*, or "servers". At each step, a new request vertex is designated as a *left-side vertex* or "client", appears and must be immediately matched to an available right-hand side vertex. The goal is to minimize the total cost of the matching.

It is easily seen that no online algorithm can be competitive if the edge-weights are allowed to be arbitrary, and hence a natural restriction is to consider the case when the edge-weights correspond to distances in a *metric space* and hence satisfy the triangle inequality. We call this problem the *online metric matching problem* . The problem arises naturally in many settings: for example, consider k fire stations, each of which can handle exactly one fire; when a new fire starts, an available fire station must be assigned to it immediately. Both Khuller, Mitchell and Vazirani [14], and Kalyanasundaram and Pruhs [9] gave $2k-1$ competitive deterministic algorithms for the online metric matching problem (and showed that no better deterministic algorithm is possible even for the star graph). Moreover, Kalyanasundaram and Pruhs also showed that the natural greedy algorithm that matches a request to its closest available point is $(2^k - 1)$-competitive,

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 522–533, 2007.

and this bound is tight. The online metric matching problem was subsequently studied for special metric spaces, and also in models with resource augmentation; see Section 1.2 for more details and several other lines of related work.

It is not difficult to give a tight $\Theta(\log k)$-competitive *randomized* solution to the online metric matching problem on the star graph, which is also the bad example in the deterministic case (see, e.g.,Section 3.1 for both upper and lower bounds). Hence, a natural question is whether randomization could help obtain an exponential improvement for general metric spaces. In a recent breakthrough, Meyerson, Nanavati and Poplawski [19] give a randomized algorithm with an $O(\log^3 k)$-competitive ratio for general metrics; this is the first algorithm with a performance sublinear in k for general metric spaces. Their approach is to first use results on approximating general metrics by tree metrics [2,3,6] to obtain an online metric matching problem instance on a class of trees called $O(\log k)$-HSTs—these are trees where the edge lengths increase by a factor of $O(\log k)$ as one goes from the leaves to the root. Then, they solve the online metric matching problem on these HSTs with a competitive ratio of $O(\log k)$; since they lose $O(\log^2 k)$ in the reduction to the HSTs, the ultimate competitive ratio obtained is $O(\log^3 k)$. In the rest of this paper, we refer to the algorithm from [19] as the *MNP Algorithm*.

One natural approach to improve the competitive ratio is to show that the MNP Algorithm also works on α-HSTs for $\alpha = O(1)$; this would immediately remove one of the logarithmic terms from the competitive ratio. However, [19] sketch an example where running MNP on an α-HST with $\alpha = o(\log k)$ would result in an extremely poor competitive ratio; We discuss this issue further in Section 3.2.

1.1 Our Results

Given that the MNP algorithm cannot be directly improved, we devise a new algorithm that proves our main technical result:

Theorem 1 (Upper Bound for 2-HSTs). *There is an $O(\log k)$-competitive algorithm for the online metric matching problem problem on 2-HSTs.*

Using standard results on approximating general metric spaces by HSTs (see Section 2), the above theorem immediately implies the following result on arbitrary metric spaces.

Corollary 1 (Upper Bound for Arbitrary Metrics). *There is an $O(\log^2 k)$-competitive algorithm for the online metric matching problem problem on arbitrary metric spaces.*

Our Techniques. Let us briefly discuss our techniques, as the proof of Theorem 1 requires a few conceptual steps. The first step shows that the natural greedy offline algorithm that repeatedly matches the closest (request,server) pair is optimal for HSTs. (This is not immediate, since the greedy algorithm is known to be bad even for a set of points on the line [21].) This analysis of the greedy algorithm allows us to lower bound the optimal cost of an instance.

The next step shows that we can imagine working in a more flexible model, where (some) server reassignments are permitted. In particular, if client c is previously assigned to server s (incurring a cost of $d(c, s)$) and a client c' arrives, we are allowed to assign c' to s, and to reassign c to some s' incurring an additional cost of $d(c, s') + d(c', s))$, as long as the new server (i.e., s') for c is no closer to it than the old one (i.e., s). Given

an online algorithm that works in this (restricted) reassignment model, we show how to get an online algorithm in the (no reassignment) original model with no greater cost. Finally, we give an online algorithm in this restricted-reassignment model which terminates with the greedy assignment, and where the total expected cost incurred during all the reassignments is at most $O(\log k)$ times the greedy cost, giving us the theorem.

1.2 Related Work

Apart from the initial work of Khuller, Mitchell and Vazirani [14], and Kalyanasundaram and Pruhs [9] on the online metric matching problem, there are several bodies of work related to our paper.

Special Metrics. The online metric matching problem has been studied for the case of special metrics as well. For example, the case of a line metric arises naturally at ski shops that have skis of various heights. As the skiers arrive one-by-one, the goal is to match skiers to skis such that the total mismatch between the length of the desired skis and the actual skis is minimized. The line case also generalizes the well-known "cow-path" problem [1]. Koutsoupias and Nanavati [15] showed that the Generalized Work Function algorithm of [10] is not constant competitive for the line metric. Fuchs et al. [7] showed a lower bound of 9.001 for the line metric, which implies that it is strictly harder than the cowpath problem. Kalyanasundaram and Pruhs [11] also considered the problem for general metrics when the adversary is allowed only half as many servers as the online servers. Recently, Chung, Pruhs and Uthaisombut [5] considered the case when the online algorithm is allowed *one extra server* at each point where the adversary has a server: somewhat surprisingly, they gave a deterministic algorithm that achieves a competitive ratio of polylog(k). Many variations of the problem have also been investigated: e.g., the minimum online bottleneck matching, the maximization version of the problem, and the case where the points are uniformly distributed on a disk in the Euclidean plane; for these, we refer the reader to the survey by Kalyanasundaram and Pruhs [10] and the references therein.

The k-Server Problem. The online metric matching problem is similar to the well-studied *k-server problem* introduced by Manasse, McGeoch and Sleator [17]: indeed, it can be viewed as a restricted case of the k-server problem where the location of the servers is fixed and not allowed to change. The methods used to give lower and upper bounds for the online matching problem, particularly on the uniform metric, are closely related to the results for k-server. Moreover, techniques used for the k-server problems, most notably work-function algorithms, have also been studied for the online metric matching problem [16,15]. Throughout this paper, we will adopt the k-server view of the problem: given an underlying metric space and k "servers" with fixed locations, find the minimum cost matching between the servers and the requests arriving online.

Offline Heuristics. The *metric case* of the matching problem has also been studied offline in the context of finding fast and simple heuristics. Reingold and Tarjan [21] showed that greedily matching the closest (request, server) pair, and recursing on the remaining instance gave an $O(k^{\log \frac{3}{2}})$-approximation to the min-cost matching, and that this bound was tight. A more sophisticated algorithm achieving an approximation bound of $O(\log k)$ was given by Plaisted [20]. Finally, Goemans and Williamson [8]

constructed a primal-dual algorithm that achieves an $2 - 2/n$-approximation for the minimum-weight perfect matching problem.

The Online Bipartite Matching Problem. A different (and equally natural) version of the bipartite matching problem was proposed by Karp, Vazirani and Vazirani [13]. In their version of the problem, the right side of the bipartite graph is given in advance and the vertices on the left side (along with their incident edges) are revealed sequentially. Upon arrival, each vertex on the left-hand side must be matched to a right-hand side vertex (if possible); moreover, these decisions are irrevocable. Karp et al. considered the *unweighted* case of the problem, where the goal is to match as many vertices as possible, and gave an optimal $(1 - 1/e)$-competitive randomized algorithm. Note that in this version of the problem triangle inequality does not hold. A generalization to the online b-matching was considered in [12], and several extensions have recently been considered in the context of allocating ad-auctions in electronic markets [18,4].

2 Preliminaries

The *metric matching problem* is formally defined as follows. We are given a metric space (V, d). In addition, we are given a set of *requests* $R \subseteq V$ where $|R| = k$, and a set of *servers* $S \subseteq V$ with $|S| = k$; the sets R and S do not have to be disjoint. The objective is to find a minimum cost bipartite matching between the requests in R and the servers in S. In the *online* version of the problem, we know in advance the metric space and the server set S; however, the set R of requests arrive one-by-one in an online fashion. Upon arrival, a request must be immediately and irrevocably matched to some unmatched server in S; changing the assignment of a previously-matched request or server is not allowed. Let $R = \{r_1, r_2, \ldots, r_k\}$ be the set of requests according to their arrival order and let $S = \{s_1, s_2, \ldots, s_k\}$ be the set of servers.

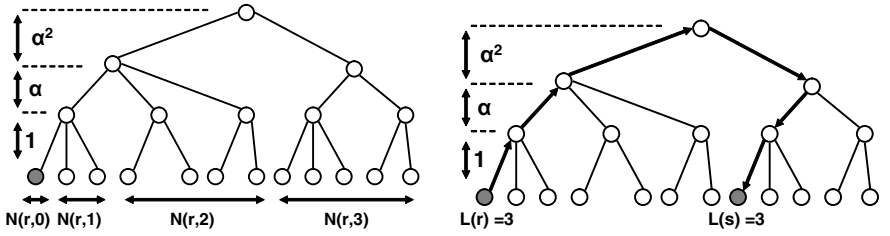
Hierarchically Well-Separated Trees (HSTs). Our results, as well as the previous results of [19], use a special type of tree metrics called *HSTs*. (See Figure 1 for an illustration).

Definition 1 (HSTs). *Given a parameter $\alpha \geq 1$, an α-Hierarchically Well-Separated Tree (α-HST) is a rooted tree $T = (V, E)$ along with a length function d on the edges which satisfies the following properties:*

1. *For each node v, all the children of v are at the same distance from v.*
2. *For any node v, if $p(v)$ is the parent of v and $c(v)$ is any child of v, then $d(p(v), v) = \alpha \cdot d(v, c(v))$.*
3. *Each leaf has the same distance to its parent.*

We view an α-HST as a leveled tree, where all the leaves are at the same level, and the edge-lengths increase geometrically by a factor of α as we go up the tree from the leaves to the root.

Let $\mathcal{H}$ be an α-HST with n leaves. For each leaf node i (that can be either request node or a server node), let $T(i, \ell)$ be the set of leaf nodes that are in the sub-tree of height ℓ that contains node i. Let $N(i, \ell)$ be the leaf nodes that are in the sub-tree $T(i, \ell)$ and not in sub-tree $T(i, \ell - 1)$. The distance of node i from each node in $N(i, \ell)$ is exactly $2\frac{(\alpha^\ell - 1)}{\alpha - 1}$. It is easy to verify that for each node i, the sets $N(i, \ell)$ induce a partition of



(a) (b)

Fig. 1. Figure (a) shows an α-HST. For a request r it illustrates the set of nodes $N(r, \cdot)$. If request r is matched to server s as in Figure (b), then $L(r) = L(s) = 3$.

the servers with respect to the node i. Given a matching in which request r is matched with server s we define $L(r)$ to be the level ℓ for which $s \in N(r, \ell)$. Similarly, let $L(s)$ be the level for which $r \in N(s, \ell)$. If server s is not matched yet we define $L(s) = \infty$. (The definitions are illustrated in Figure 1).

Reducing from General Metrics to HSTs. The results of [6] imply that given any metric (V, d) on n points, there is a probability distribution on α-HST's with the following properties: **(a)** For each HST T in the support of the distribution, the leaves of T correspond to the nodes of V, and the distance in the tree $d_T(u, v) \geq d(u, v)$ for all $u, v \in V$, and **(b)** the expected distance $E[d_T(u, v)]$ between two nodes $u, v \in V$ is at most $O(\alpha \log n)d(u, v)$, where the expectation is taken over the random choice of the HST T. Furthermore, one can sample from this distribution in polynomial time.

Using this result, a β-competitive randomized algorithm for our problem on an α-HST directly implies an $O(\alpha\beta \log n)$-competitive randomized algorithm on the original metric. Note that a-priori, the number of nodes n in the metric space can be much larger than k. However, we can still replace $\log n$ by $\log k$ following an idea of Meyerson et al. [19]: we construct the HST only for the submetric induced on the k server nodes. Now, whenever a request arrives at some point p, we pretend that it has arrived at the server $s(p)$ closest to it, and handle it accordingly. Using the triangle inequality and the fact that $d(p, s(p))$ summed over all requests is a lower bound on the optimum solution, can change the competitive ratio by at most a constant factor.

3 Previous Algorithms

In this section we describe several basic results and arguments, which will crucially be used in the rest of the paper. We also describe the randomized greedy algorithm which is the basis of the previous $O(\log^3 k)$ result of [19]. We show that the analysis is in fact tight, and hence a different algorithm is necessary to obtain a better result.

3.1 The Uniform Metric

We begin by describing the $\Theta(\log k)$ lower and upper bounds for the uniform metric. Recall that the uniform metric consists of a set of points such that any two points are at unit distance from each other. For the lower bound, consider the uniform metric on

$k+1$ points labeled $0, 1, \ldots, k$, and suppose that points $1, 2, \ldots, k$ (i.e., all except 0) contain one server each. The adversary places the first request at point 0. At each subsequent step for the next $k-1$ steps, the adversary requests a point that has not been requested thus far and is most likely to have a matched server. Note that just before the i^{th} request is made (for $i \geq 2$), there are $(k-i+2)$ unrequested points each containing a server, and one of these servers has already been matched. Hence the probability that the server at the requested point is already matched is at least $1/(k-i+2)$. Summing over all the requests, the expected cost incurred by any online algorithm is at least $1 + \sum_{i=2}^{k} 1/(k-i+2) = H_k = \Omega(\log k)$. The offline algorithm on the other hand only incurs a cost of 1.

We now show a matching upper bound. For a uniform metric (on, say $n \geq k$ points), each request is either collocated with a server, or else it is at unit distance from it. Consider the following algorithm inspired by the above lower bound: when a request arrives, if there a collocated server still available, we match the request to it; otherwise we choose an available server at unit distance uniformly at random and match the request to it. Consider an instance where u of the arriving requests are not collocated with a server (i.e., lie outside the set S). Clearly, the optimum offline algorithm has cost u; moreover, the online algorithm also pays cost for each such non-collocated request. Now consider the requests that arrive at points collocated with a server. Just before the i^{th} such request arrives, suppose there have already been u_i non-collocated requests. The crucial observation is that all the previous $i-1$ server locations where collocated requests arrived have already been matched (either to some collocated request or one of the u_i requests). Morever, for each of the remaining $k-i+1$ server locations, each of them is unavailable with probability exactly $u_i/(k-i+1)$, which is at most $u/(k-i+1)$. Hence, the i^{th} request has an expected cost of at most $u/(k-i+1)$; summing up over the non-collocated and the collocated requests, the total expected cost incurred is

$$u + \sum_{i=1}^{k-u} \frac{u}{k-i+1} \leq u + uH_k = O(u \log k).$$

3.2 The Randomized Greedy Algorithm

Meyerson et al. [19] considered the following simple randomized greedy algorithm: whenever a request arrives, match it to the nearest unmatched server. If there are several unmatched servers that qualify, choose one of these unmatched servers uniformly at random. In general metrics this approach can yield a competitive ratio as bad as $2^k - 1$ [9]. However, this algorithm performs quite well on α-HST metrics with large enough value of α. Specifically, Meyerson et al. analysed this algorithm on an α-HST with $\alpha \geq 2\ln k + 1$, and proved it is $O(\log k)$-competitive. This immediately implied an $O(\alpha \log^2 k) = O(\log^3 k)$ competitive algorithm for general metrics.

An appealing approach to improve the competitive factor is to try to remove the requirement that $\alpha \geq 2\ln k + 1$. Meyerson et al. showed that $\alpha = \Omega(\log k)$ is necessary for the randomized greedy algorithm to work. Specifically, it is possible to prove the following Lemma:

Lemma 1 ([19]). *For any constant ℓ, there exists an ℓ level α-HST and an input instance with optimal cost $O(\alpha^{\ell-1})$, such that the MNP Algorithm incurs a cost of $\Omega(\sum_{i=0}^{\ell-1} (\log k)^{i+1} \alpha^{\ell-i-1})$.*

We remark that it is easily checked that if $\alpha = o(\log k)$, then the online cost is substantially larger than the offline cost. A close inspection of the lower bound example reveals that the lower levels have a disproportionately higher contribution to the online cost as compared with the offline algorithm. The main problem is that the MNP algorithm incurs too much cost in the lower levels until it realizes that there are no available servers in a subtree and that it needs to find a server outside the subtree. The lower bound example motivates a different approach that is used by our new constructed algorithm.

4 An $O(\log k)$ Algorithm for 2-HST's

In this section we present a simple online algorithm which is $O(\log k)$-competitive on an α-HST, for any constant value of α. For simplicity we set $\alpha = 2$. Our algorithm has three conceptual steps. First, in Section 4.1 we describe a simple offline algorithm that computes an optimal matching on an HST. In Section 4.2 we define a restricted reassignment online model which is easier to handle. We then prove that we can obtain an online algorithm with *no reassignments* from any online algorithm in the restricted reassignment model without compromising the competitive ratio. Finally, in Section 4.3, we design a simple online algorithm in the restricted reassignment model and prove that it is $O(\log k)$-competitive.

4.1 An Offline Algorithm

In this section we design a simple **offline** algorithm that computes an optimal solution on an HST metric. This algorithm is essentially the greedy approach that matches the closest request-server pair, and then recurses on the remaining instance. Reingold and Tarjan [21] proved that this approach leads to a very poor $\theta(k^{\log \frac{3}{2}})$-approximation in general metrics. However, we show here that this greedy approach leads to an optimal solution in the special case of HST metrics.

1. Let R and S be the current sets of unmatched requests and unmatched servers, respectively. Initially, R and S contain all requests and servers.
2. Iterate on the levels from level $\ell = 0$ and up until the highest level.
3. Iterate on the requests $r \in R$ in **any order**.
4. For each request r, if $N(r, \ell) \cap S \neq \emptyset$ match r to **any server** in $N(r, \ell) \cap S$ and remove r and s from R and S, respectively. Otherwise, continue to the next request in R.

We refer to this algorithm as the *Generic Algorithm*, since it considers the requests in arbitrary order, and the server it chooses for each request r from the set $N(r, \ell) \cap S$ is also arbitrary. Thus, the algorithm is *flexible* with respect to these choices, meaning that the output of the algorithm is not unique. This property of the algorithm will be very important in the sequel. We say that a matching M is *feasible* with respect to the Generic Algorithm if there exists a run of the algorithm that can generate M. The next Lemma proves that the algorithm outputs an optimal matching on any HST metric.

Lemma 2. *The Generic Algorithm generates an optimal matching on an HST metric.*

Proof. Consider a subtree T_i rooted at a node i at height ℓ. Let $R(i)$ and $S(i)$ be the number of requests and the number of servers in T_i respectively. Clearly, any solution must match at least $E(i) = \max(0, R(i) - S(i))$ requests belonging to T_i to servers that lie outside T_i. These requests must incur a cost of $2\alpha^\ell \cdot E(i)$ when going up from level ℓ to level $\ell + 1$ (node i's parent) and then coming down from level $\ell + 1$ back to level ℓ. Thus, the optimum cost OPT(T) on the whole HST is at least $\sum_{i \in T} 2\alpha^{\ell(i)} E(i)$, where the summation is over all nodes i of T, and $\ell(i)$ denotes the level of i.

Now consider the behavior of our Generic Algorithm. When it first considers level ℓ, in each subtree T_i rooted at level ℓ, no server in T_i can be occupied by a request outside T_i. Moreover, at Step (3), the unsatisfied requests in T_i are matched within T_i as much as possible. Thus, exactly $E(i)$ requests remain unmatched in T_i after level ℓ is processed, and hence, by the same reasoning as above, the matching produced has cost exactly $\sum_{i \in T} 2\alpha^{\ell(i)} E(i)$.

4.2 A Restricted Reassignment Online Model

In this section we define a different online model and prove that it suffices to design a competitive online algorithm for this modified model. We refer to the new model as the *restricted reassignment online model*; as the name suggests, this model allows some reassignment of previously arrived requests. Specifically, in the new model we are allowed to reassign a previously matched request r_p with the following restriction: if, currently, r_p is matched to a server belonging to $N(r_p, \ell)$ for some value ℓ, then the algorithm is allowed to reassign r_p only to a server belonging to $N(r_p, \ell')$, where $\ell' \geq \ell$. The online algorithm in the restricted reassignment model pays the cost of **all** reassignments performed, and not just the cost of the final matching computed.

We claim that any online algorithm in the restricted reassignment model can be transformed to an online algorithm in the original model (where no reassignments are allowed) with no additional cost. This is done by a very simple method. First, we can assume without loss of generality that the algorithm in the new model is lazy and does not reassign requests unnecessarily. That is, it only reassigns a request if a currently occupied server by it must be used to match another request. Consider a move of the algorithm, where r is matched to s_1 that was previously matched to r_1. Request r_1 is then matched to server s_2 which was previously matched to r_2, and so on, until request r_t which was previously matched to s_t is reassigned to a vacant server s_{t+1}. The change in the matching is viewed in the following:

$$\underbrace{s_1 \leftarrow r_1, s_2 \leftarrow r_2, \ldots, s_t \leftarrow r_t}_{\text{The original matching}} \Rightarrow \underbrace{r \rightarrow s_1, r_1 \rightarrow s_2, \ldots, r_{t-1} \rightarrow r_t, r_t \rightarrow s_{t+1}}_{\text{The matching after the reassignment process}}$$

The cost of reassigning the requests in the new model is $d(r, s_1) + \sum_{i=1}^{t} d(r_i, s_{i+1})$. An algorithm in an online model with no reassignments would simulate this move by simply matching r directly to s_{t+1}, paying a cost of $d(r, s_{t+1})$. The following lemma shows that the total cost of this algorithm with no reassignments is no more than the cost incurred in the restricted reassignment model.

Lemma 3. *In any iteration of the algorithm:* $d(r, s_{t+1}) \leq d(r, s_1) + \sum_{i=1}^{t} d(r_i, s_{i+1})$

Proof. The claim follows directly from the restrictions on the reassignments in the new model. Assume that r_i was matched to a server s_i in level ℓ. Thus, r_i and s_i both belong to the tree $T(r_i, \ell)$. By the restriction on the reassignments, r_i cannot be reassigned to a server in $T(r_i, \ell-1)$. Thus, the path from r_i to server s_{i+1} passes through the root of the tree $T(r_i, \ell)$. Therefore, the path from s_i (that is, in the sub-tree $T(r_i, \ell)$) to s_{i+1} is at most $d(r_i, s_{i+1})$. Thus, we get that for any $i \in \{1, 2, \ldots, t\}$, $d(s_i, s_{i+1}) \leq d(r_i, s_{i+1})$. Using the triangle inequality we get that:

$$d(r, s_{t+1}) \leq d(r, s_1) + \sum_{i=1}^{t} d(s_i, s_{i+1}) \leq d(r, s_1) + \sum_{i=1}^{t-1} d(r_i, s_{i+1})$$

4.3 An $O(\log k)$-Competitive Algorithm in the Restricted Reassignment Model

In this section we present an $O(\log k)$-competitive algorithm for the online metric matching in the restricted reassignment model. The algorithm is as follows:

Initially, set $L(s) = \infty$ for all servers in S.
When request r arrives, set $L(r) = 0$:

1. Find the lowest level $\ell \geq L(r)$ in which there exists a server $s \in N(r, \ell) \cap S$ such that $L(s) > \ell$.
2. Choose uniformly at random a server s among the servers in $N(r, \ell) \cap S$ for which $L(s) > \ell$.
3. Match r to s (and set $L(r) = L(s) = \ell$).
4. If s was previously matched to another request r', then reassign r' using the same procedure (return to Step (1) with r').

Note that Step (1) above ensures that for each request r, its level $L(r)$ can only increase during the execution of the algorithm. Thus, the algorithm satisfies the requirement of the restricted reassignment model. Suppose the arrival of a new request r causes the reassignment of requests $r_1, r_2, \ldots, r_p$. Then, the number of reassignments is at most the height of the HST, since for all $i < p$, $L(r_i) < L(r_{i+1})$. Also, by the condition in Step (1) above, the level $L(s)$ of each server s can only decrease during the execution of the algorithm. The next lemma shows that the final solution produced by the algorithm is optimal (without taking into account the cost of reassignments).

Lemma 4. *The final matching produced by the online algorithm is optimal.*

Proof. We show that the solution produced by the online algorithm is feasible with respect to the offline Generic Algorithm of Section 4.1. Thus, by Lemma 2 the solution is optimal.

Consider the final matching produced upon termination of the online algorithm. Let $R_i \subseteq R$ be the set of requests such that $L(r) = i$. We show how to obtain the final online solution using the Generic Algorithm. In the first round we match all the servers in R_0 to the servers that are used by the online algorithm. In the second round we match the requests in R_1 to the servers used by the online algorithm, and so on. It suffices to show that this corresponds to a feasible run of the Generic Algorithm. To this end, it is enough to show that for any i, after having matched the subset R_i, we cannot match any request $r \in R \setminus \left(\bigcup_{j=0}^{i} R_i\right)$ to servers in level i with respect to r.

Assuming that this is not true, then after having matched the requests in R_i, there still exists an unmatched request r that can be matched with a server $s \in N(r,i)$. Consider the online iteration in which request r had $L(r) \leq i$ and was matched (or reassigned) to a server in a level higher than i. Such an iteration must exist as $L(r)$ starts from zero and the request r is eventually matched to a level higher than i. Moreover, in this iteration, $L(s) > i$, since upon termination of the algorithm $L(s) > i$ (none of the requests from the subset $\bigcup_{j=1}^{i} R_j$ is matched to s), and the level $L(s)$ of each server can only decrease during the execution of the algorithm. Therefore, at this iteration, request r could have been matched with a server in level i (with respect to r). This contradicts the fact that in that iteration request r chose to be matched with a server with a level strictly larger than i.

Lemma 5. *The expected cost of the reassignments of a request r which is matched upon termination of the online algorithm to a server s (at distance $d(r,s)$) is $O(\log k)d(r,s)$.*

Proof. Consider a request r which is matched upon termination of the online algorithm to a server s at level $L(r) = L(s)$. During the execution of the algorithm, $L(r)$ is monotonically non-decreasing. Consider a level ℓ, $0 \leq \ell \leq L(r)$. We prove that the expected number of times r is matched (or reassigned) to servers in $N(r,\ell)$ is $O(\log N(r,\ell)) = O(\log k)$. The intuition is the following. During the execution of the algorithm, request r can cause a reassignment of request r' only if r' is matched to a server s' which is strictly closer to r (i.e. $s' \in N(\ell, r)$, $s' \in N(\ell', r')$ and $\ell < \ell'$). In this case we say that r is *stronger* than r'. Request r' then chooses a new server which is at least at the same distance from r' as s' and is not occupied by any request which is at least as strong as r'. The set of servers satisfying the latter condition is the feasible set for r' and its size is monotonically (strictly) decreasing over the execution of the algorithm. The main observation is that r' always chooses uniformly at random a new server from the set of feasible servers. Thus, the probability that the next request which is stronger than r' will cause another reassignment of r' is at most the inverse of the size of the set of feasible servers for r'. This gives us the harmonic number as an upper bound on the expected number of reassignments, similarly to the uniform metric case.

Formally, fix any sequence of requests. Assume that at some time during the online algorithm request r is matched to a server in $N(r,\ell)$ (if there is no such iteration then we are done). Next, define the set $W \subseteq N(r,\ell)$ of feasible matchings for request r to be the set of servers in $N(r,\ell)$ for which $L(s) > \ell$. The size of W can only decrease throughout the execution of the algorithm. Next, consider the arrival order of requests which are at least as strong as r and are matched to servers in W (until either request r is matched to a higher level or until the end of the execution). If some request having the same strength as r arrives the probability that it causes a reassignment of r is zero. Otherwise, since in each reassignment, the request r chooses uniformly at random a server among the remaining servers in W, the probability that such a request causes r to be reassigned is at most $\frac{1}{|W|}$, where $|W|$ is the current size of the set of feasible servers for r. In either case the size of W decreases by 1 after each such arrival. Since initially $|W| \leq |N(r,\ell)|$ (similarly to the uniform metric), it follows that the expected number of reassignments until r is matched to a higher level, or is matched to its final server, is $O(H_{N(r,\ell)}) = O(\log |N(r,\ell)|) = O(\log k)$.

Since the metric is a 2-HST, the cost of each reassignment of request r in level i costs $2(2^i - 1)$. Therefore, the total expected cost of reassigning request r is at most: $O(\log k) \sum_{i=0}^{L(r)} 2(2^i - 1) = O(\log k) 2^{L(r)} = O(\log k) d(r, s)$, where the last inequality follows since $d(r, s) = 2(2^{L(r)} - 1)$.

Theorem 2. *The online algorithm is $O(\log k)$ competitive on HST's.*

Proof. By linearity of expectation and Lemma 5 the total expected cost of the algorithm is $O(\log k)$ times the final cost of the solution of the online algorithm. By Lemma 4 the final solution is optimal and thus the total expected cost of the algorithm is actually $O(\log k)$ times the optimum. Finally, by Lemma 3 we can translate this algorithm easily to the model with no reassignments without increasing the cost.

5 Conclusion

In this paper, we designed an algorithm for the online metric matching problem which is $O(\log k)$-competitive on 2-HST's, and thus $O(\log^2 k)$-competitive for general metrics. The main open question is to design an algorithm with competitive ratio $O(\log k)$, or to improve the known lower bound. The analysis of such an algorithm cannot proceed along the same lines as we pursue, since approximating general metrics by an HST incurs a loss of $\Omega(\log k)$ in the worst case, and moreover, there is an $\Omega(\log k)$ lower bound for the online metric matching problem even on HSTs.

Interestingly, the claims in Sections 4.2 and 4.3 can be extended easily to general metrics, which implies that our algorithm is $O(\log k)$-competitive for those metrics on which the greedy approach of Section 4.1 generates a constant factor approximation. However, since the lower bound of Reingold and Tarjan [21] shows the existence of metrics for which the greedy approach produces an $\Omega(k^{\log \frac{3}{2}})$-approximate solution, applying our algorithm directly on such metrics might be as bad as $\Omega(k^{\log \frac{3}{2}} \log k)$-competitive. Nonetheless, a possible direction to obtain an $O(\log k)$-competitive algorithm for general metrics might be to combine our techniques in Sections 4.2 and 4.3 with a different offline heuristic, that results in an $O(1)$-approximation. It is somewhat strange to look for an $O(1)$-approximation for a problem that is in polynomially solvable. However, such an algorithm that takes advantage of the metric properties of the graph might be a key ingredient. A possible starting point can be the "hyper-greedy" heuristic of Supowit, Plaisted, and Reingold [23] that achieves an $O(\log k)$-approximation to the offline metric matching problem, or the $2 - 2/n$ primal dual approximation algorithm of Goemans and Williamson [8].

References

1. Baeza-Yates, R.A., Culberson, J.C., Rawlins, G.J: Searching in the plane. Information and Computation 106(2), 234–252 (1993)
2. Bartal, Y.: Probabilistic approximations of metric spaces and its algorithmic applications. In: IEEE Symposium on Foundations of Computer Science, pp. 184–193. IEEE Computer Society Press, Los Alamitos (1996)

3. Bartal, Y.: On approximating arbitrary metrics by tree metrics. In: STOC: ACM Symposium on Theory of Computing (STOC), ACM Press, New York (1998)
4. SBuchbinder, N., Jain, K., Naor, J.: The adauctions problem and extensions. In: ESA (to appear 2007)
5. Chung, C., Pruhs, K., Uthaisombut, P.: The online transportation problem: On the exponential boost of one extra server. Under submission
6. Fakcharoenphol, J., Rao, S., Talwar, K.: A tight bound on approximating arbitrary metrics by tree metrics. In: STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, pp. 448–455. ACM Press, New York (2003)
7. Fuchs, B., Hochstattler, W., Kern, W.: Online matching on a line. Theoretical Computer Science, 251–264 (2005)
8. Goemans, M.X., Williamson, D.P.: A General Approximation Technique for Constrained Forest Problems. SIAM Journal on Computing 24, 296–317 (1995)
9. Kalyanasundaram, B., Pruhs, K.: Online weighted matching. J. Algorithms 14(3), 478–488 (1993)
10. Kalyanasundaram, B., Pruhs, K.: Online network optimization problems, 1998. In: Fiat, A., Woeginger, G. (eds.) Online Algorithms. LNCS, vol. 1442, Springer, Heidelberg (1998)
11. Kalyanasundaram, B., Pruhs, K.: The online transportation problem. SIAM J. Discrete Math. 13(3), 370–383 (2000)
12. Kalyanasundaram, B., Pruhs, K.: An optimal deterministic algorithm for online b -matching. Theoretical Computer Science 233(1–2), 319–325 (2000)
13. Karp, R.M., Vazirani, U.V., Vazirani, V.V.: An optimal algorithm for online bipartite matching. In: In Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, pp. 352–358. ACM Press, New York (1990)
14. Khuller, S., Mitchell, S.G., Vazirani, V.V.: On-line algorithms for weighted bipartite matching and stable marriages. Theor. Comput. Sci. 127(2), 255–267 (1994)
15. Koutsoupias, E., Nanavati, A.: The online matching problem on a line. In: Solis-Oba, R., Jansen, K. (eds.) WAOA 2003. LNCS, vol. 2909, pp. 179–191. Springer, Heidelberg (2004)
16. Koutsoupias, E., Papadimitriou, C.: On the k-server conjecture. Journal of the ACM 42(5), 971–983 (1995)
17. Manasse, M.S., McGeoch, L.A., Sleator, D.D.: Competitive algorithms for online problems. In: STOC: ACM Symposium on Theory of Computing, pp. 322–333. ACM Press, New York (1988)
18. Mehta, A., Saberi, A., Vazirani, U.V., Vazirani, V.V.: Adwords and generalized on-line matching. In: IEEE Symposium on Foundations of Computer Science, pp. 264–273. IEEE Computer Society Press, Los Alamitos (2005)
19. Meyerson, A., Nanavati, A., Poplawski, L.: Randomized online algorithms for minimum metric bipartite matching. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 954–959. ACM Press, New York (2006)
20. Plaisted, D.A.: Heuristic matching for graphs satisfying the triangle inequality. J. Algorithms 5(2), 163–179 (1984)
21. Reingold, E.M., Tarjan, R.E.: On a greedy heuristic for complete matching. SIAM Journal on Computing 10(4), 676–681 (1981)
22. Schrijver, A.: Combinatorial optimization. Polyhedra and efficiency. Algorithms and Combinatorics, vol. 24. Springer, Berlin (2003)
23. Supowit, K.J., Reingold, E.M., Plaisted, D.A.: The travelling salesman problem and minimum matching in the unit square. SIAM J. Comput. 12(1), 144–156 (1983)

To Fill or Not to Fill: The Gas Station Problem*
(Extended Abstract)

Samir Khuller, Azarakhsh Malekian, and Julián Mestre

Department of Computer Science.
University of Maryland, College Park, MD 20742, USA
{samir,malekian,jmestre}@cs.umd.edu

Abstract. In this paper we study several routing problems that generalize shortest paths and the Traveling Salesman Problem. We consider a more general model that incorporates the actual cost in terms of gas prices. We have a vehicle with a given tank capacity. We assume that at each vertex gas may be purchased at a certain price. The objective is to find the cheapest route to go from s to t, or the cheapest tour visiting a given set of locations. Surprisingly, the problem of find the cheapest way to go from s to t can be solved in polynomial time and is not NP-complete. For most other versions however, the problem is NP-complete and we develop polynomial time approximation algorithms for these versions.

1 Introduction

Optimization problems related to computing the shortest (or cheapest) tour visiting a set of locations, or that of computing the shortest path between a pair of locations are pervasive in Computer Science and Operations Research. Typically, the measures that we optimize are in terms of "distance" traveled, or time spent (or in some cases, a combination of the two). There are literally thousands of papers dealing with problems related to shortest-path and tour problems.

In this paper, we consider a more general model that incorporates the actual cost in terms of *gas prices*. We have a vehicle with a given tank capacity of U. In fact, we will assume that U is the distance the vehicle may travel on a full tank of gas (this can easily be obtained by taking the product of the tank size and the mileage per gas unit of the vehicle). Moreover, we may assume that we start with some given amount of gas μ ($\leq U$) in the tank. We assume that at each vertex v gas may be purchased at a price of $c(v)$. This price is the cost of gas per mile. For example if gas costs \$3.40 per gallon and the vehicle can travel for 17 miles per gallon, then the cost per mile is 20 cents.

At each gas station we may fill up some amount of gas to "extend" the range of the vehicle by a certain amount. Moreover, since gas prices vary, the cost depends on where we purchase gas from.

* Research supported by NSF grant CCF-0430650. Full version available at http://www.cs.umd.edu/∼samir/grant/esa07.ps

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 534–545, 2007.

In addition to fluctuating gas prices, there is significant variance in the price of gas between gas stations in different areas. For example, in the Washington DC area alone, the variance in gas prices between gas stations in different areas (on the same day) can be by as much as 20%. Due to different state taxes, gas prices in adjacent states also vary. Finally, one may ask: why do we expect such information to be available? In fact, there are a collection of web sites [1,2] that currently list gas prices in an area specified by zip code. So it is reasonable to assume that information about gas prices is available. What we are interested in are algorithms that will let us compute solutions to some basic problems, given this information.

In this general framework, we are interested in a collection of basic questions.

1. (The gas station problem) Given a start node s and a target node t, how do we go from s to t in the cheapest possible way if we start at s with μ_s amount of gas? In addition we consider the variation in which we are willing to stop to get gas at most Δ times[1]. Another generalization we study is the sequence gas station problem. Here, we want to find the cheapest route that visits a set of p locations in a specified order (for example by a delivery vehicle).
2. (The fixed-path gas station problem) An interesting special case is when we fix the path along which we would like to travel. Our goal is to find an optimal set of refill stops along the path.
3. (The uniform cost tour gas station problem) Given a collection of cities T, and a set of gas stations S at which we are willing to purchase gas, find the shortest tour that visits T. We have to ensure that we never run out of gas. Clearly this problem generalizes the Traveling Salesman Problem. The problem gets more interesting when $S \neq T$, and we address this case. This models the situation when a large transportation company has a deal with a certain gas company, and their vehicles may fill up gas at any station of this company at a pre-negotiated price. Here we assume that gas prices are the same at each gas station. This could also model a situation where some gas stations with very high prices are simply dropped from consideration, and the set S is simply the set of gas stations that we are willing to use.
4. (The tour gas station problem) This is the same as the previous problem, except that the prices at different stations can vary.

Of all the above problems, only the tour problems are NP-hard. For the first two we develop polynomial time algorithms, and for the tour problems we develop approximation algorithms.

We now give a short summary of the results in the paper:

1. (The gas station problem) For the basic gas station problem, our algorithm runs in time $O(\Delta n^2 \log n)$ and computes an optimal solution. If we want

[1] This restriction makes sense, because in some situations where the gas prices are decreasing as we approach our destination, the cheapest solution may involve an arbitrarily large number of stops, since we only fill up enough gas to make it to a cheaper station further down the path.

to visit a sequence of p cities we can find an optimal solution in time $O(\Delta(np)^2 \log(np))$. In addition, we develop a second algorithm for the all-pairs version that runs in time $O(n^3\Delta^2)$. This method is better than repeating the fixed-destination algorithm n times when $\Delta < \log n$.

2. (The fixed-path gas station problem) For the fixed-path version with an unbounded number of stops, we develop a fast $O(n \log n)$ time algorithm. Due to space constraints this is described in the full version of the paper.
3. (The uniform cost tour gas station problem) Since this problem is NP-hard, we focus on polynomial time approximation algorithms. We assume that every city has a gas station within a distance of $\alpha\frac{U}{2}$ for some $\alpha < 1$. This assumption is reasonable since in any case, every city has to have a gas station within distance $\frac{U}{2}$, otherwise there is no way to visit it. A similar assumption is made in the work on distance constrained vehicle routing problem [13]. We develop an approximation algorithm with an approximation factor of $\frac{3}{2}(\frac{1+\alpha}{1-\alpha})$. We also consider a special case, namely when there is only one gas station. This is the same as having a central depot, and requiring the vehicle to return to the depot after traveling a maximum distance of U. For this special case, we develop an algorithm with factor $O(\ln \frac{1}{1-\alpha})$ and this improves the bound of $\frac{3}{2(1-\alpha)}$ given by Li et al. [13] for the distance constrained vehicle routing problem.
4. (The tour gas station problem) For the tour problem with arbitrary prices, we can use the following scheme: sort all the gas prices in non-decreasing order $c_1 \leq c_2 \leq \ldots c_n$. Now guess a range of prices $[c_i \ldots c_j]$ one is willing to pay, and let $\beta_{ij} = \frac{c_j}{c_i}$. Let S_{ij} include all the gas stations v such that $c_i \leq c(v) \leq c_j$. We can run the algorithm for the *uniform cost tour gas station problem* with set S_{ij} and cities T. This will yield a tour $\mathcal{T}^{[i,j]}$. We observe that the cost of the tour $\mathcal{T}^{[i,j]}$ is at most $O(\frac{\beta_{ij}}{1-\alpha})$ times the cost of an optimal solution, since its possible that we always pay a factor β_{ij} more than the optimal solution, at each station where we fill gas. Taking the best solution over all $O(n^2)$ possible choices gives a valid solution to the tour gas station problem.

1.1 Related Work

The problems of computing shortest paths and the shortest TSP tour are clearly the most relevant ones here and are widely studied, and discussed in several books [12,17].

One closely related problem is the Orienteering problem [4,5,10,7]. In this problem the goal is to compute a path of a fixed length L that visits as many locations as possible, starting from a specified vertex. For this problem, a factor 3 approximation has been given recently by Bansal et al. [6]. (In fact, they can fix the starting and ending vertices.) This algorithm is used as subroutine for developing a bicriteria bound for Deadline TSP. By using the 3 approximation for the Orienteering problem, we develop an $O(\log |T|)$ approximation for the single gas station tour problem. This is not surprising, since we would like to cover all the locations by finding walks of length at most U.

There has been some recent work by Nagarajan and Ravi [16] on minimum vehicle routing that is closely related to the single gas station tour problem. In this problem, a designated root vertex (depot) and a deadline D are given and the goal is to use the minimum number of vehicles from the root so that each location is met by at least one of the vehicles, and each vehicle traverses length at most D. (In their definition, vehicles do not have to go back to the root.) They give a 4-approximation for the case where locations are in a tree and an $O(\log D)$ approximation for graphs with integer weights.

Another closely related piece of work is by Arkin et al. [3] where tree and tour covers of bounded length are computed. What makes their problem easier is that there is no specified root node, or a set of gas stations one of which should be included in any bounded length tree or tour. Several pieces of work deal with vehicle routing problems [14,15,9] with multiple vehicles, where the objective is to bound the total cost of the solution, or to minimize the longest tour. However these problems are significantly easier to develop approximation algorithms for.

2 The Gas Station Problem

The input to our problem consists of a complete graph $G = (V, E)$ with edge lengths $d : E \to R^+$, gas costs $c : V \to R^+$ and a tank capacity U. (Equivalently, if we are not given a complete graph we can define d_{uv} to be the distance between u and v in G.) Our goal is to go from a source s to a destination t in the cheapest possible way using at most Δ stops to fill gas. For ease of exposition we concentrate on the case where we start from s with an empty tank. The case in which we start with μ_s units of gas can be reduced to the former as follows. Add a new node s' such that $d_{s's} = U - \mu_s$ and $c(s') = 0$. The problem of starting from s with μ_s units of gas and that of starting from s' with an empty tank using one additional stop are equivalent.

We would also like to note that our strategy yields a solution where the gas tank will be empty when one reaches a location where gas can be filled cheaply. In practice, this is not safe and one might run out of gas (for example if one gets stuck in traffic). For that reason we suggest defining U to be *smaller* than the actual tank capacity so that we always have some "reserve" capacity.

In this section we develop an $O(\Delta n^2 \log n)$ time algorithm for the gas station problem. In addition, when $\Delta = n$ we show how to solve the problem in $O(n^3)$ time for general graphs, and $O(n \log n)$ time for the case where G is a fixed path.

One interesting generalization of the problem is the *sequence gas station problem* where we are given a sequence $s_1, s_2, \ldots, s_p$ of vertices that we must visit in the specified order. This variant can be reduced to the s-t version in an appropriately defined graph.

2.1 The Gas Station Problem Using Δ Stops

We will solve the gas station problem using the following dynamic program (DP) formulation:

$A(u, q, g)$ = Minimum cost of going from u to t using q refill stops, starting with g units of gas. We consider u to be one of the q stops.

The main difficulty in dealing with the problem stems from the fact that, in principle, we need to consider every value of $g \in [0, U]$. One way to avoid this is to discretize the values g can take. Unfortunately this only yields a pseudo-polynomial time algorithm. To get around this we need to take a closer look at the structure of the optimal solution.

Lemma 1. *Let $s = u_1, u_2, \ldots, u_l$ be the refill stops of an optimal solution using at most Δ stops. The following is an optimal strategy for deciding how much gas to fill at each stop: At u_l fill just enough to reach t with an empty tank; for $j < l$*

i) If $c(u_j) < c(u_{j+1})$ then at u_j fill up the tank.
ii) If $c(u_j) \geq c(u_{j+1})$ then at u_j fill just enough gas to reach u_{j+1}.

Proof. If $c(u_j) < c(u_{j+1})$ and the optimal solution does not fill up at u_j then we can increase the amount filled at u_j and decrease the amount filled at u_{j+1}. This improves the cost of the solution, which contradicts the optimality assumption. Similarly, if $c(u_j) \geq c(u_{j+1})$ then we can decrease the amount filled at u_j and increase the amount filled at u_{j+1} (without increasing the overall cost of the solution) until the condition is met. □

Consider a refill stop $u \neq s$ in the optimal solution. Let w be the stop right before u. Lemma 1 implies that if $c(w) > c(u)$, we reach u with an empty tank, otherwise we reach u with $U - d_{wu}$ gas. Therefore, in our DP formulation we need to keep track of at most n different values of gas for u. Let $GV(u)$ be the set of such values, namely

$$GV(u) = \{U - d_{wu} \,|\, w \in V \text{ and } c(w) < c(u) \text{ and } d_{wu} \leq U\} \cup \{0\}$$

The following recurrence allows us to compute $A(u, q, g)$ for any $g \in GV(u)$:

$$A(u, 1, g) = \begin{cases} (d_{ut} - g)\, c(u) & \text{if } g \leq d_{ut} \leq U \\ \infty & \text{otherwise} \end{cases}$$

$$A(u, q, g) = \min_{\substack{v \text{ s.t.} \\ d_{uv} \leq U}} \left\{ \begin{array}{ll} A(v, q-1, 0) + (d_{uv} - g)\, c(u) & \mid c(v) \leq c(u) \wedge g \leq d_{uv} \\ A(v, q-1, U - d_{uv}) + (U - g)\, c(u) & \mid c(v) > c(u) \end{array} \right\}$$

The cost of the optimal solution is $\min_{1 \leq l \leq \Delta} A(s, l, 0)$. The naive way of filling the table takes $O(\Delta n^3)$ time. However, this can be done more efficiently.

Theorem 1. *There is an $O(\Delta n^2 \log n)$ time algorithm for the gas station problem with Δ stops.*

Instead of spending $O(n)$ time computing a single entry of the table, we spend $O(\log n)$ amortized time per entry. More precisely, for fixed $u \in V$ and $1 < q \leq \Delta$ we show how to compute all entries of the form $A(u, q, *)$ in $O(n \log n)$ time using entries of the form $A(*, q-1, *)$. Theorem 1 follows immediately.

The DP recursion for $A(u, q, g)$ finds the minimum, over all v such that $d_{uv} \leq U$, of terms that corresponds to the cost of going from u to t through v. Split

each of these terms into two parts based on whether they depend on g or not. Thus we have an independent part, which is either $A(v, q-1, 0) + d_{uv}\, c(u)$ or $A(v, q-1, U-d_{uv}) + Uc(u)$; and a dependent part, $-g\, c(u)$.

Our procedure begins by sorting the independent part of every term. Note that the minimum of these corresponds to the entry for $g = 0$. As we increase g, the terms decrease uniformly. Thus, to compute the table entry for $g > 0$ just subtract $g\, c(u)$ from the smallest independent part available. The only caveat is that the term corresponding to a vertex v such that $c(v) \leq c(u)$ should not be considered any more once $g > d_{uv}$, we say such a term *expires* after $g > d_{uv}$. Since the independent terms are sorted, once the smallest independent term expires we can walk down the sorted list to find the next vertex which has not yet expired. The procedure is dominated by the time spent sorting the independent terms which takes $O(n \log n)$ time.

Theorem 2. *When $\Delta = n$ the problem can be solved in $O(n^3)$ time.*

We can reduce the problem to a shortest path question on a new graph H. The vertices of H are pairs (u, g), where $u \in V$ and $g \in GV(u)$. The edges of H and their weight $w(\cdot)$ are defined by the DP recurrence. Namely, for every $u, v \in V$ and $g \in GV(u)$ such that $d_{uv} \leq U$ we have $w\big((u,q),(v,0)\big) = (d_{uv} - g)\, c(u)$ if $c(v) \leq c(u)$ and $g \leq d_{uv}$, or $w\big((u,q),(v,U-d_{uv}\big) = (U-g)\, c(u)$ if $c(v) > c(u)$.

Our objective is to find a shortest path from $(s, 0)$ to $(t, 0)$. Note that H has at most n^2 vertices and at most n^3 edges. Using Dijkstra's algorithm [8] the theorem follows.

2.2 Faster Algorithm for the All-Pairs Version

Consider the case in which we wish to solve the problem for all starting nodes i, with μ_i amount of gas in the tank initially. Using the method described in the previous section, we get a running time of $O(n^3 \Delta \log n)$ since we run the algorithm for each possible destination. We will show that for $\Delta < \log n$ we can improve this and get a bound of $O(n^3 \Delta^2)$.

Add new nodes i' such that $d_{i'i} = U - \mu_i$ and $c(i') = 0$. If we start at i with μ_i units of gas, it is the same as starting from i' where gas is free. We fill up the tank to capacity U, and then by the time we reach i we will have exactly μ_i units of gas in the tank. (Since gas is free at any node i' in any optimal solution we fill up the tank to capacity U). This will use one extra stop.

We define $B(i, h, p)$ as the minimum cost solution to go from i to h (destination), with p stops to get gas, given that we start with an empty tank at i. Since we start with an empty tank, we have to fill up gas at the starting point (and this is included as one of the stops). Clearly, we will also reach h (destination) with an empty tank, assuming that there is no trivial solution, such as one that arrives at the destination with no fill-ups on the way.

Our goal is to compute $B(i', h, \Delta + 1)$ which is a minimum cost solution to go from i' to h with at most Δ stops in-between. Note that the first fill-up is the one that takes place at node i', after that we stop at most Δ times.

We will now show how to compute $B(i,h,p)$. There are two options:

- If the gas price at the first stop after i (e.g. k) is cheaper than $c(i)$ then we will reach that station with an empty tank after filling d_{ik} units of gas at i (as long as $d_{ik} \leq U$):

$$B(i,h,p) = B(k,h,p-1) + d_{ik}c(i)$$

- If the first place where the cost of gas decreases from the previous stop is the $q+1^{st}$ stop and the price is in increasing order in the first q stops then

$$B(i,h,p) = C(i,k,q) + B(k,h,p-q)$$

  We define $C(i,k,q)$ as the minimum cost way of going from i to k with at most q stops to get gas, such that we start at i with an empty tank (and get gas at i, which counts as a stop) and finally reach k with an empty tank. In addition, the price of gas in intermediate stations is in increasing order except for the last stop.

We define $B(h,h,p) = 0$. For $i \neq h$ let $B(i,h,1) = c(i)\, d_{ih}$ if $d_{ih} \leq U$, and $B(i,h,1) = \infty$ otherwise. In general:

$$B(i,h,p) = \min \left\{ \min_{\substack{1 \leq k \leq n \\ 1 < q \leq p}} C(i,k,q) + B(k,h,p-q), \min_{\substack{1 \leq k \leq n \\ \text{s.t.} d_{ik} \leq U}} B(k,h,p-1) + d_{ik}\, c(i) \right\}$$

If we are able to compute $C(i,k,q)$ efficiently then $B(i,h,p)$ can be computed. There are $n^2\Delta$ states in the dynamic program, and each one can be computed in time $O(n\Delta)$. This yields a running time of $O(n^3\Delta^2)$. We will see that the time required to compute $C(i,k,q)$ is $O(n^3\Delta)$ for all relevant choices of i,k,q.

Suppose that in going from i to k we stop at $i_1 = i, \ldots, i_q, i_{q+1} = k$. Note that $c(i_1) \leq c(i_2) \leq \ldots \leq c(i_q)$, however $c(i_q) > c(i_{q+1})$. In fact, at i_1 we will get U amount of gas. When we reach i_j for $1 < j < q$, we will get $d_{i_{j-1}i_j}$ units of gas (the amount that we consumed since the previous fill-up) at a cost of $c(i_j)$ per unit of gas. The amount of gas we will get at i_q is just enough to reach k with an empty tank. Now we can see that the total cost is equal to $Uc(i_1) + d_{i_1 i_2}c(i_2) + \ldots + d_{i_{q-2}i_{q-1}}c(i_{q-1}) + (d_{i_{q-1}i_q} + d_{i_q k} - U)c(i_q)$. Note that the last term is not negative, since we could not reach k from i_{q-1} even with a full tank at i_{q-1}, without stopping to get a small amount of gas.

We compute $C(i,k,q)$ as follows. First note that if $d_{ik} \leq U$ then the answer is $d_{ik}c(i)$. Otherwise we build a directed graph $G' = (V \cup V_D, E \cup E_D)$, where V is the set of vertices, and $V_D = \{i' | i \in V\}$.

We define E: add a directed edge from $i \in V$ to j for each vertex $j \in V \setminus \{i\}$ such that $d_{ij} \leq U$ and $c(i) \leq c(j)$. The weight of this edge is $d_{ij}c(j)$.

We define E_D as follows: add a directed edge from each $j \in V$ to k' for each vertex $k' \in V_D \setminus \{j'\}$ such that $U < d_{jk} \leq 2U$. The weight of this edge is

$$\min \left\{ (d_{jz} + d_{zk} - U)c(z) \,|\, c(j), c(k) < c(z) \text{ and } d_{jz}, d_{zk} \leq U \right\}$$

Now we can express $C(i,k,q)$ as $Sp(i,k',q) + Uc(i)$ where $Sp(i,k',q)$ is the shortest path from i to k' in the graph G' using at most q edges.

To see why it is true, we can see that for any given order of stops between i and k (where the gas price is in increasing order in consecutive stops), the minimum cost is equal to the weight of the path in G' that starts from i, goes to the second stop in the given order (e.g., i_2) and then traverses the vertices of V in the same order and from the second last stop goes to k'. It is also possible that $q = 2$ and the path goes directly from $i = i_1$ to k in this case, and i_2 is the choice for z that achieves the minimum cost for the edge (i,k').

For any given path P in G' between i and k', if the weight of the path is W_P we can find a feasible plan for filling the tank at the stations so that the cost is equal to $W_P + Uc(i)$. It is enough to fill up the tank at the stations that are in the path, except the last one in which the tank is filled to only the required level to reach k. We can conclude that $C(i,k,q)$ is equal to $Sp(i,k',q) + Uc(i)$.

The running time for finding the shortest path between all pairs of nodes with different number of stops (at most Δ) can be computed in $O(n^3\Delta)$ by dynamic programming [11]. If we precompute $C(i,k,q)$ the running time for computing $B(i',h,\Delta+1)$ is $O(n^3\Delta^2)$ assuming we start at i with μ_i amount of gas. So in general the running time is $O(n^3\Delta^2)$.

3 The Uniform Cost Tour Gas Station Problem

In this section we study a variant of the gas station problem where we must visit a set of cities T in arbitrary order. We consider the case where gas costs the same at every gas station, but some cities may not have a gas station.

More formally, the input to our problem consists of a complete undirected graph $G = (V,E)$ with edge lengths $d : E \to R^+$, a set of cities $T \subseteq V$, a set of gas stations $S \subseteq V$, and tank capacity U for our vehicle. The objective is to find a minimum length tour that visits all cities in T, and possibly some gas stations in S. We are allowed to visit a location multiple times if necessary. We require any segment of the tour of length U to contain at least one gas station, this ensures we never run out of gas. We call this the *uniform cost tour gas station problem*. We assume that we start with an empty tank at a gas-station.

The problem is NP-hard as it generalizes the well-known traveling salesman problem: just set the tank capacity to the largest distance between any two cities and let $T = S$. In fact, there is a closer connection between the two problems: If every city has a gas station, i.e., $T \subseteq S$, we can reduce the gas station problem to the TSP. Consider a TSP instance on T under metric $\ell : T \times T \to R^+$, where ℓ_{xy} is the minimum cost of going between cities x and y starting with an empty tank (this can be computed by standard techniques). Since the cost of gas is the same everywhere, a TSP tour can be turned into a driving plan that visits all cities with the same cost and vice-versa. Let OPT denote an optimal solution, and $c(OPT)$ its cost.

As mentioned earlier, we can use the algorithm for the uniform cost case to derive an approximation algorithm for the general case by paying a factor β

in the approximation ratio. Here β is the ratio of the maximum price that an optimal solution pays for buying a unit of gas, to the minimum price it pays for buying a unit of gas (in practice this ranges from 1 to 1.2).

Unfortunately this reduction to the TSP breaks down when cities are not guaranteed to have a gas station. Consider going from x to y, where x does not have a gas station. The distance between x and y will depend on how much gas we have at x, which in turn depends on which city was visited before x and what route we took to get there.

An interesting case of the tour gas station problem is that of an instance with a single gas station. This is also known as the *distance constrained vehicle routing problem* and was studied by Li et al. [13] who gave a $\frac{3}{2(1-\alpha)}$ approximation algorithm, where the distance from the gas station to the most distant city is $\alpha\frac{U}{2}$, for some $\alpha < 1$. We improve this by providing an $O(\log \frac{1}{1-\alpha})$ approximation algorithm Without making any assumptions on α we show that a greedy algorithm that finds bounded length tours visiting the most cities at a time is a $O(\log |T|)$-factor approximation. The proof of these claims appear in the full version of this paper.

For the general case we make the assumption that every city has a gas station at distance at most $\alpha\frac{U}{2}$. This assumption is reasonable, because if a city has no gas station within distance $\frac{U}{2}$, there is no way to visit it. We show a $\frac{3(1+\alpha)}{2(1-\alpha)}$ approximation for this problem. Note that when $\alpha = 0$, this gives the same bound as the Christofides method for the TSP.

3.1 The Tour Gas Station Problem

For each city $x \in T$ let $g(x) \in S$ be the closest gas station to x, and let d_x be the distance from x to $g(x)$. We assume that every city has a gas station at distance at most $\alpha\frac{U}{2}$; in order words, $d_x \leq \alpha\frac{U}{2}$ for all $x \in T$.

Recall that it is assumed that the price of the gas is the same at all the gas stations. We define a new distance function for the distance between each pair of cities. The distance ℓ is defined as follows: For each pair of cities x and y, ℓ_{xy} is the length of the shortest traversal to go from x to y starting with $U - d_x$ amount of gas and reaching y with d_y amount of gas. If $d_{xy} \leq U - d_x - d_y$ then we can go directly from x to y, and $\ell_{xy} = d_{xy}$. Otherwise, we can compute this as follows. Create a graph whose vertex set is S, the set of gas stations. To this graph add x and y. We now add edges from x to all gas stations within distance $U - d_x$ from x. Similarly we add edges from y to all gas stations within distance $U - d_y$ to y. Between all pairs of gas stations, we add an edge if the distance between the pair of gas stations is at most U. All edges have length equal to the distance between their end points. The length of the shortest path in this graph from x to y will be ℓ_{xy}. Note that the shortest path (in general) will start at x and then go through a series of gas stations before reaching y. This path yields a valid plan to drive from x to y without running out of gas, once we reach x with $U - d_x$ units of gas. When we reach y, we have enough gas to go to g_y. Also note that $\ell_{xy} = \ell_{yx}$ since the path is essentially "reversible".

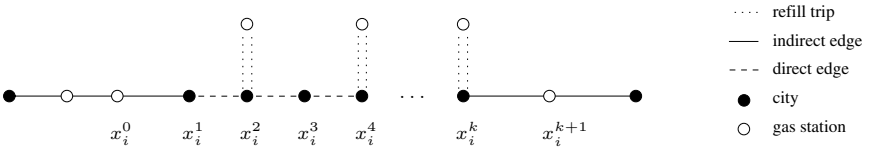


Fig. 1. Decomposition of the solution into strands

We assume here that all distances are Euclidean. Note that from x, we can only go to B and not A since we start from x with $U - d_x$ units of gas. From B, we cannot go to D since the distance between B and D is more than U, even though the path through D to y would be shorter. From C we go to E since going through F will give a longer path, since from F we cannot go to y directly.

Note that the function ℓ may not satisfy triangle inequality. To see this, suppose we have three cities x, y, z. Let $d_{xy} = d_{yz} = \frac{U}{2}$. Let $d_x = d_y = d_z = \frac{U}{4}$ and $d_{xz} = U$. We first observe that $\ell_{xy} = \ell_{yz} = \frac{U}{2}$. However, if we compute ℓ_{xz}, we cannot go from x to z directly since we only have $\frac{3}{4}U$ units of gas when we start at x and need to reach z with $\frac{U}{4}$ units of gas. So we have to visit g_y along the way, and thus $\ell_{xz} = \frac{3}{2}U$.

The algorithm is as follows:

1. Create a new graph G', with a vertex for each city. For each pair of cities x, y compute ℓ_{xy} as shown earlier.
2. Find the minimum spanning tree in (G', ℓ). Also find a minimum weight perfect matching M on the odd degree vertices in the MST. Combine the MST and M to find an Euler tour $\mathcal{T}$.
3. Start traversing the Eulerian tour. Add refill trips whenever needed. (Details on this follow).

It can be shown that the total length of the MST is less than the optimal solution cost. Suppose $x_1, \ldots, x_n$ is the order in which the optimal solution visits the cities. Clearly, the cost of going from x_i to x_{i+1} in the optimal solution is at least $\ell_{x_i x_{i+1}}$. Since the collection of edges (x_i, x_{i+1}) forms a spanning tree, we can be conclude that the weight of the $\ell(\text{MST}) \leq c(OPT)$. Next we show that the cost of M is at most $\frac{c(OPT)}{2}$. Suppose the odd degree vertices are in the optimal solution in the order $o_1, \ldots, o_k$. We can see that $\ell_{o_i o_{i+1}}$ is at most equal to the distance we travel in the optimal solution to go from o_i to o_{i+1}. So the cost of minimum weighted matching on the odd degree vertices is at most $\frac{c(OPT)}{2}$. So the total cost of the Eulerian tour $\mathcal{T}$ is at most $\frac{3c(OPT)}{2}$.

Now we need to transform the Eulerian tour into a feasible plan. First, every edge (x, y) in $\mathcal{T}$ is replaced with the actual plan to drive from x to y that we found when computing ℓ_{xy}. If $d_{xy} \leq U - d_x - d_y$ the plan is simply to go straight from x to y, we call these *direct edges*. Otherwise the plan must involve stopping along the way in one or more gas stations, we call these *indirect edges*. Notice that the cost of this plan is exactly that of the Eulerian tour $\mathcal{T}$. Unfortunately, as we will see below this plan need not be feasible.

Define a strand, to be a sequence of consecutive cities in the tour connected by direct edges. If a city is connected with two indirect edges, then it forms a strand by itself. Suppose the i^{th} strand has cities $x_i^1, \ldots, x_i^k$. To this we add x_i^0 (x_i^{k+1}), the last (first) gas station in the indirect edge connecting x_i^1 (x_i^k) with the rest of the tour. Each strand now starts and ends with a gas station. We can view the tour as a decomposition into strands as shown in Fig. 1. Note that if the distance between x_i^0 and x_i^{k+1} is more than U the overall plan is not feasible. To fix we add for every city a refill trip to its closest gas station and then greedily try to remove them, while maintaining feasibility, until we get a minimal set of refill trips. Let us bound the extra cost these trips incur.

Lemma 2. *Let L_i be the length of the ith strand. Then the total distance traveled on the refill trips of cities in the strand is at most $\frac{2\alpha}{1-\alpha} L_i$.*

Proof. Assume there are q_i refill trips in this strand. Label the cities with refill trips to their nearest gas stations $x_i^{j_1}, \ldots, x_i^{j_{q_i}}$. Also label x_i^0 as $x_i^{j_0}$ and x_i^k as $x_i^{j_{q_i+1}}$. Note that $\ell(\mathcal{T}(x_i^{j_p}, x_i^{j_{p+2}})) \geq (1-\alpha)U$ (otherwise the refill trip at $x_i^{j_{p+1}}$ can be dropped). This gives us:

$$2L_i \;>\; \sum_{0 \leq p \leq q_i - 1} \ell(\mathcal{T}(x_i^{j_p}, x_i^{j_{p+2}})) \;\geq\; q_i(1-\alpha)U \;\Longrightarrow\; q_i \leq \frac{2L_i}{(1-\alpha)U}$$

The length of each refill trip no more than αU. Therefore, the total length of the refill trips is at most $\alpha U q_i$, and the lemma follows. □

The cost of the solution is the total length of the strands (which is the length of the tour) plus the total cost of the refill trips. (Note that without loss of generality we can assume that our tour always starts from a gas station. For the case with only direct edges, there is exactly one strand, starting and ending at the first city with the gas station).

In other words, the total cost of the solution is:

$$\ell(\mathcal{T}) + \sum_i \alpha U q_i \leq \left(1 + \frac{2\alpha}{1-\alpha}\right) \ell(\mathcal{T}) \leq \left(\frac{1+\alpha}{1-\alpha}\right) \frac{3}{2}\, c(\mathrm{OPT}).$$

Theorem 3. *There is a $\frac{3}{2}\frac{(1+\alpha)}{(1-\alpha)}$-approximation for the tour gas station problem.*

4 Conclusion

Current problems of interest are to explore improvements in the approximation factors for the special cases of Euclidean metrics, and planar graphs. In addition we would also like to develop faster algorithms for the single source and destination case, perhaps at the cost of sacrificing optimality of the solution.

References

1. http://www.gasbuddy.com/
2. http://www.aaa.com/
3. Arkin, E.M., Hassin, R., Levin, A.: Approximations for minimum and min-max vehicle routing problems. Journal of Algorithms 59(1), 1–18 (2006)
4. Arkin, E.M., Mitchell, J.S.B., Narasimhan, G.: Resource-constrained geometric network optimization. In: Proceedings of the 14th Annual Symposium on Computational Geometry (SoCG), pp. 307–316 (1998)
5. Awerbuch, B., Azar, Y., Blum, A., Vempala, S.: New approximation guarantees for minimum-weight k-trees and prize-collecting salesmen. SIAM Journal on Computing 28(1), 254–262 (1998)
6. Bansal, N., Blum, A., Chawla, S., Meyerson, A.: Approximation algorithms for deadline-TSP and vehicle routing with time-windows. In: Proceedings of the 36th annual ACM symposium on Theory of computing (STOC), pp. 166–174 (2004)
7. Blum, A., Chawla, S., Karger, D.R., Lane, T., Meyerson, A., Minkoff, M.: Approximation algorithms for orienteering and discounted-reward TSP. In: Proceedings of the 44rd Annual IEEE Symposium on Foundations of Computer Science (FOCS), p. 46 (2003)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. M.I.T. Press and McGraw-Hill (2001)
9. Frederickson, G.N., Hecht, M.S., Kim, C.E.: Approximation algorithms for some routing problems. SIAM Journal on Computing 7(2), 178–193 (1978)
10. Golden, B.L., Levy, L., Vohra, R.: The orienteering problem. Naval Research Logistics 34, 307–318 (1987)
11. Lawler, E.L.: Combinatorial Optimization: Networks and Matroids. Dover Publications, Mineola, NY (2001)
12. Lawler, E.L., Lenstra, J.K., Kan, A.H.G.R., Shmoys, D.B.: The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. John Wiley & Sons, Chichester (1985)
13. Li, C.-L., Simchi-Levi, D., Desrochers, M.: On the distance constrained vehicle routing problem. Operations Research 40(4), 790–799 (1992)
14. Haimovich, A.G.R.K.M.: Bounds and heuristics for capacitated routing problems. Mathematics of Operations Research 10(4), 527–542 (1985)
15. Haimovich, L.S.M., Rinnoooy Kan, A.G.: Analysis of heuristics for vehicle routing problems. Vehicle Routing: Methods and Studies, pp. 47–61 (1988)
16. Nagarajan, V., Ravi, R.: Minimum vehicle routing with a common deadline. In: Díaz, J., Jansen, K., Rolim, J.D.P., Zwick, U. (eds.) APPROX 2006 and RANDOM 2006. LNCS, vol. 4110, pp. 212–223. Springer, Heidelberg (2006)
17. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization. Dover Publications, Inc, Mineola, NY (1998)

Online Bandwidth Allocation*

Michal Forišek[1], Branislav Katreniak[1], Jana Katreniaková[1], Rastislav Královič[1], Richard Královič[1,2], Vladimír Koutný[1], Dana Pardubská[1], Tomáš Plachetka[1], and Branislav Rovan[1]

[1] Dept. of Computer Science, Comenius University, Mlynská dolina, 84248 Bratislava, Slovakia
[2] Dept. of Computer Science, ETH Zürich, Switzerland

Abstract. The paper investigates a version of the resource allocation problem arising in the wireless networking, namely in the OVSF code reallocation process. In this setting a complete binary tree of a given height n is considered, together with a sequence of requests which have to be served in an online manner. The requests are of two types: an insertion request requires to allocate a complete subtree of a given height, and a deletion request frees a given allocated subtree. In order to serve an insertion request it might be necessary to move some already allocated subtrees to other locations in order to free a large enough subtree. We are interested in the worst case average number of such reallocations needed to serve a request.

In [4] the authors delivered bounds on the competitive ratio of online algorithm solving this problem, and showed that the ratio is between 1.5 and $O(n)$. We partially answer their question about the exact value by giving an $O(1)$-competitive online algorithm.

In [3], authors use the same model in the context of memory management systems, and analyze the number of reallocations needed to serve a request in the worst case. In this setting, our result is a corresponding amortized analysis.

1 Introduction and Motivation

Universal Mobile Telecommunications System (UMTS) is one of the third-generation (3G) mobile phone technologies that uses W-CDMA as the underlying standard, and is standardized by the 3GPP [8]. The W-CDMA (Wideband Code Division Multiple Access) is a wideband spread-spectrum 3G mobile telecommunication air interface that utilizes code division multiple access. The main idea behind the W-CDMA is to use physical properties of interference: if two transmitted signals at a point are in phase, they will "add up" to give twice the amplitude of each signal, but if they are out of phase, they will "subtract" and give a signal that is the difference of the amplitudes. Hence, the signal received by a particular station is the sum (component-wise) of the respective transmitted vectors of all senders in the area. In the W-CDMA, every sender s is given

* The research has been supported by grant APVV-0433-06 and VEGA 1/3106/06.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 546–557, 2007.

a *chip code* $\mathbf{v}$. Let us represent the data to be sent by a vector of ± 1. When s wants to send a *data vector* $\mathbf{d} = (d_1, \ldots, d_n)$, $d_i \in \{1, -1\}$, it sends instead a sequence $d_1 \cdot \mathbf{v}, d_2 \cdot \mathbf{v}, \ldots, d_n \cdot \mathbf{v}$, i.e. n-times the chip code modified by the data. For example, consider a sender with a chip code $(1, -1)$ that wants to send data $(1, -1, 1)$; then the actually transmitted signal is $(1, -1, -1, 1, 1, -1)$. The signal received by a station is then a sum of all transmitted signals. Clearly, if the chip codes are orthogonal, it is possible to uniquely decode all the signals.

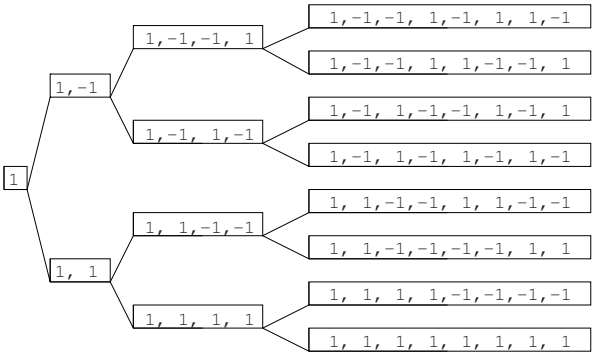


Fig. 1. An OVSF tree

One commonly used method of implementing the chip code allocation is Orthogonal Variable Spreading Factor Codes (OVSF). Consider a complete binary tree, where the root is labeled by (1) , the left son of a vertex with label α is labeled (α, α) and the right son is labeled $(\alpha, -\alpha)$ (see Figure 1).

If a sender station enters the system, it is given a chip code from the tree in such a way that there is at most one assigned code from each root-to-leaf path. It can be shown [7, 1] that this construction fulfills the orthogonality property even with codes of different lengths.

Clearly, a code at level l in the tree has length 2^l, and a sender using this code will use a fraction of $1/2^l$ of the overall bandwidth. When users enter the system, they request a code of a given length. It is irrelevant which particular code is assigned to which user, the length is the only thing that matters. When users connect to and disconnect from a given base station, i.e. request and release codes, the tree can become fragmented. It may happen that no code at the requested level is available, even though there is enough bandwidth (see Figure 2).

This problem can be solved by changing the chip codes of some already registered users, i.e. reallocating the vertices of the tree. Since the cost of a reallocation dominates this operation, the number of reallocations should be kept minimal. In [4] the authors considered the problem of minimizing the number of reallocations over a given sequence of requests and showed that it is NP-hard to generate an optimal allocation schedule. In this paper, we show that the online

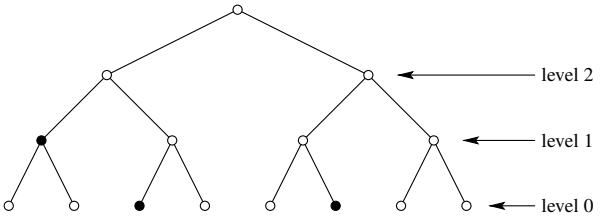


Fig. 2. No code at level 2 can be allocated although there is enough free bandwidth. Full circles represent allocated vectors.

version proposed in [4] can be solved in amortized complexity $O(1)$ reallocations per request. Due to space restrictions some technical parts have been omitted from this paper, and can be found in the technical report [5].

2 Problem Definition

Consider a complete binary tree $T = (V, E)$ of height n with leaves at level 0 and the root at level n. A *request vector* is a vector $\mathbf{r} = (r_0, \ldots, r_n) \in \mathbb{N}^{n+1}$, where r_i represents the number of users that request a code at level i. A *code assignment* of a particular request vector $\mathbf{r}$ is a subset of vertices $F \subset V$, such that every path from a leaf to the root contains at most one vertex from F, and there are exactly r_i vertices at level i in F.

The input consists of a sequence of requests of two types: *insertion* and *deletion*. Let $\mathbf{r}_t$ be the request vector and F_t its corresponding code assignment after t requests. The algorithm has to process the next request in the following way:

A) **insertion request** at a vertex at level i.
The algorithm must output a new code assignment F_{t+1} satisfying the request vector $\mathbf{r}_{t+1} = (r_0, \ldots, r_i + 1, \ldots, r_n)$.[1]

B') **deletion request** of a particular vertex $v \in F_t$.
Let v be at level i. The new request vector is $\mathbf{r}_{t+1} = (r_0, \ldots, r_i - 1, \ldots, r_n)$, and the new code assignment is $F_{t+1} = F_t \setminus \{v\}$.

During each step, the number of reassignments is $|F_{t+1} \setminus F_t|$. For a given sequence of requests $R_1, \ldots, R_m$, we are interested in the amortized number of reassignments per request, i.e. the quantity $\frac{1}{m} \sum_{t=0}^{m-1} |F_{t+1} \setminus F_t|$.

Note that there is no harm in allowing the reallocations within the deletion requests. In a deletion request the algorithm remembers the moves it would perform, and performs them in the next insertion request; in case of consecutive deletion requests it is enough to maintain a mapping between the actual vertices

[1] We may assume, without loss of generality, that there is enough bandwidth to satisfy each request.

in F and their "virtual" positions. This leads to a reformulation of a deletion request – instead of deleting the particular vertex v at level i, algorithm is allowed to delete *some* vertex at level i instead:

B) deletion request of a vertex at level i.
The algorithm is required to produce a new code assignment F_{t+1} satisfying the request vector $\mathbf{r}_{t+1} = (r_0, \ldots, r_i - 1, \ldots, r_n)$.

This definition bears resemblance to memory allocation problems studied in the operating systems community, in particular to the *binary buddy system* memory allocation strategy introduced in [6]. In this strategy, requests to allocate and deallocate memory blocks of sizes 2^l are served. The system maintains a list of free blocks of sizes of 2^k. When allocating a block of size 2^l in a situation where no block of this size is free, some bigger block is recursively split into two halves called *buddies*. When a block whose buddy is free is deallocated, both buddies are recursively recombined into a bigger block.

The properties of binary buddy system have been extensively studied in the literature (see e.g. [2] and references therein). However, the bulk of this research is focused on cases without reallocation of memory blocks. E.g. [2] shows how to implement the buddy system in amortized constant time per allocation/deallocation request, but in a model where reallocation is not allowed. As the restriction of not allowing reallocations makes the model substantially different, the results in [2] can not be directly applied to the model with reallocations allowed.

A binary buddy system with memory block reallocations has been studied in [3]. This paper analyzes both the number of reallocated blocks and the number of reallocated bytes per request; the analysis of the number of reallocated blocks is in fact the very same model that is used in our paper. However, only the worst case scenario of simple greedy algorithms is analyzed in [3] and presented results are quite pessimistic: $s - 1$ blocks have to be reallocated in worst case when allocating a block of size s. There are stronger results presented in [3], but all of them require a significant amount of auxiliary memory.

3 The Algorithm

In this section we propose an online algorithm that processes the sequence of insertion/deletion requests according to the rules A) and B). The goal is to keep the amortized number of reallocations per request constant. We start with some notions and notations.

If we order the leaves from left to right and label them with numbers $0, \ldots, 2^n - 1$, there is an interval of the form $I_u = \langle i2^l, (i+1)2^l - 1 \rangle$ assigned to each tree vertex u of level l. We call such an interval a *place* of level l, and say that I_u begins at position $i2^l$. For a given code assignment F, a place I_u corresponding to a vertex u is called an *empty* (or *free*) *place* if neither u nor any vertex from the subtree rooted at u is in F. For $u \in F$ the corresponding place I_u is an *occupied place*, and we say that there is a *pebble* of level l located on

I_u (or, alternatively there is a pebble of level l at position $i2^l$). Since in a code assignment F, every path from a leaf to the root contains at most one vertex, we can view the code assignment F as a sequence of disjoint places which are either empty or occupied by pebble (see Figure 3). While the free places in this decomposition are not uniquely defined, we shall overlook this ambiguity, as we will argue either about a particular place or about the overall size of free places (called also *free bandwidth*).

We say that a pebble (or place) of level l has *size* 2^l. The left (right) neighbor of a pebble is a pebble placed on the next occupied place to the left (right). Analogously we talk about a left (right) neighbor of a free place. We denote pebbles by capital letters $A, B, \ldots, X$, and their corresponding sizes by $a, b, \ldots, x$. The notion of a vertex and the corresponding place are used interchangeably.

The key idea of our algorithm is to maintain a well defined structure in the sequence of pebbles. An obvious approach would be to keep the sequence sorted – i.e. the pebbles of lower levels always preceding the pebbles of higher levels with only the smallest necessary free places between them. It is easy to see that the addition to and deletion from such a structured sequence can be done with at most $O(n)$ reallocations.[2] Unfortunately, it is also not difficult to see that there is a sequence of requests such that this approach needs amortized $O(n)$ reallocations per request (see [4]). The problem is that a too strictly defined structure needs too much "housekeeping" operations.

To overcome this problem we will keep the maintained structure less rigid. The pebbles are colored and kept so that the *black* pebbles form a sorted sequence according to the previous rule. There are several restrictions imposed on *white* pebbles. Informally, if there is a free place in the black sequence, a single white pebble can be placed at the beginning of this place. Moreover, all white pebbles form an increasing sequence. When moved, the pebble can change its color.

Formally, the structure of the sequence of pebbles is described by the following invariants.

C: Pebble A is *black*, if there is no bigger pebble before A, otherwise it is *white*.
P1: The free bandwidth before any pebble X is strictly less than x.
P2: There is always at least one black pebble between any two white pebbles.
P3: There is no white pebble between two black pebbles of the same size.

A sequence of pebbles and free places satisfying **C, P1**–**P3** will be called a *valid situation*. The key observation about valid situations is that in a valid situation it is always possible to process an insertion request without reallocations:

Lemma 1. *Consider a valid situation, and an insertion request of size* $a = 2^l$. *If there is a free bandwidth of at least* a, *then there is also a free place of size* a.

[2] The addition request is processed by placing the pebble of level l at the end of sequence of pebbles of this level. If this place is already occupied by some pebble B, B is removed and reinserted. Since there are at most n different levels, and the levels of reinserted pebbles are increasing, the whole process ends after $O(n)$ iterations. The deletion works in a similar fashion.

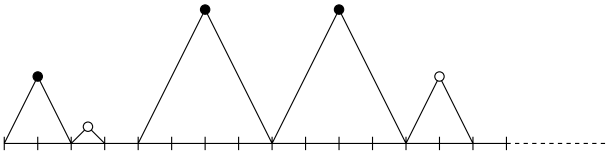


Fig. 3. A code assignment seen as a sequence of black and white pebbles

The algorithm is designed to maintain the valid situations over the whole computation, so all requests can be processed without reallocations. However, while processing a request, the intermediate situation might temporarily become not valid. The key to an effective algorithm is to develop a post-processing phase in each request that restores the validity using only a few reallocations. We show that in the case of insertion requests, a constant number of reallocations in each request is sufficient. In the deletion requests, however, a more involved accounting argument is used to show that the *average* number of reallocations per request in a worst-case execution remains constant.

Last notion connected to a structure of pebbles is that of *closing position*.

Definition 1. *The* closing position *of level l (of size x) is the position after the last black pebble of level l (of size x) if such a pebble exists. Otherwise, the closing position of level l is the position of the first pebble of higher level (size bigger than x).*

3.1 Procedure INSERT

We assume that whenever the procedure INSERT is called, it is to process a new request of size a in a valid situation in which there is a free bandwidth of size at least a. First, a free place of size a is found, and a pebble is put on this place. If the sequence is no longer valid after this operation, the algorithm reassigns a constant number of pebbles in order to restore the invariants. The procedure is listed as Algorithm 1, and its analysis is given in the following theorem:

Theorem 1. *Consider a valid situation, and an insertion request of size a. If there is a free bandwidth of size at least a, procedure* INSERT *correctly processes the request. Moreover, the output situation is valid, and only a constant number of pebbles has been reassigned within the execution.*

Sketch of proof. Here, we present an overall structure of the proof with a number of unproven claims. The complete version can be found in [5].

Consider an insertion request of size a, and suppose the invariants **C, P1**–**P3** hold. Let P be the first free place of size a; Lemma 1 ensures existence of such place. If P does not have a left neighbor (i.e. there is no other pebble present) or its left neighbor B is black, then the algorithm puts the pebble A on P and the situation remains valid.

From now on suppose that P has a white left neighbor B, and denote the potential right neighbor of P by Z. However, there exists a left neighbor C of

Algorithm 1. Procedure INSERT: inserts a pebble A of size a into a valid situation in which the free bandwidth of size $\geq a$ is guaranteed

```
 1: let P be the first free place of size a
 2: let B be the left neighbor of P
 3: if B does not exist or B is black
    then
 4:     put A on P
 5:     return
 6: end if
 7:
 8: let C be the left neighbor of B
 9: if c < a then
10:     put A on P
11:     return
12: else if c = a then
13:     remove B
14:     put A just after C
15:     put B just after A
16:     return
17: end if
18:
19: remove B
20: if a < b then
21:     rename A and B so that A
        is the bigger pebble
22: end if
23:
24: let D be the pebble at the closing
    position of size a.
25: if D is white then
26:     let E be D's right neighbor
27:     remove D, E
28:     put A at D's original position
29:     put B after A
30:     put D after B
31:     put E at B's original position
32: else if D is black then
33:     remove D
34:     put A at D's original position
35:     put B after A, put D after C
36: end if
```

B, such that C is black and $c > b$. We distinguish three sub-cases: $c < a$, $c = a$, and $c > a$. When $c < a$, the possible right neighbor Z is black and its size is bigger than a. Therefore the black pebble A might be put on P resulting in a valid situation (line 10). If $c = a$, the sequence of lines 13–16 is executed: first, white pebble B is temporarily removed. Since there has been no other pebble between A and C, and $a = c$, the place of size a immediately following C is now free and black A is placed immediately after C. Since $b < c = a$, the place of size b immediately following A is now free, so white B can be put there. Finally, if $c > a$, the algorithm temporarily removes B[3], and calls the pebble at the closing position of size a by D (there must be a pebble present). The proof is concluded by considering two final sub-cases based on whether D is white or black. In the first case, the action is depicted on Figure 4. In case of black D the situation is as follows from Figure 5. □

3.2 Procedure DELETE

The deletion request requires to remove one pebble of a specified level. In our approach, the last (rightmost) pebble A of the requested level is chosen (see Procedure DELETE, Algorithm 2). However, this action may violate the invariants **P1**–**P3**. To remedy this, the algorithm uses several iterations to "push the problem" to the right. The main "problem" in this case is the free place caused

[3] Assume w.l.o.g. that $a \geq b$, see [5].

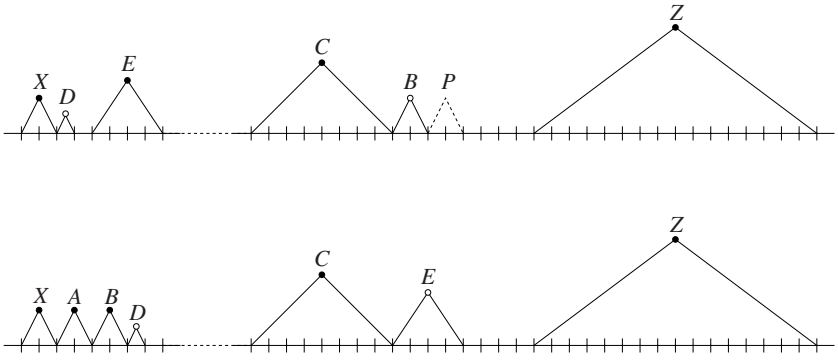


Fig. 4. An example of executing lines 26–31 of procedure INSERT

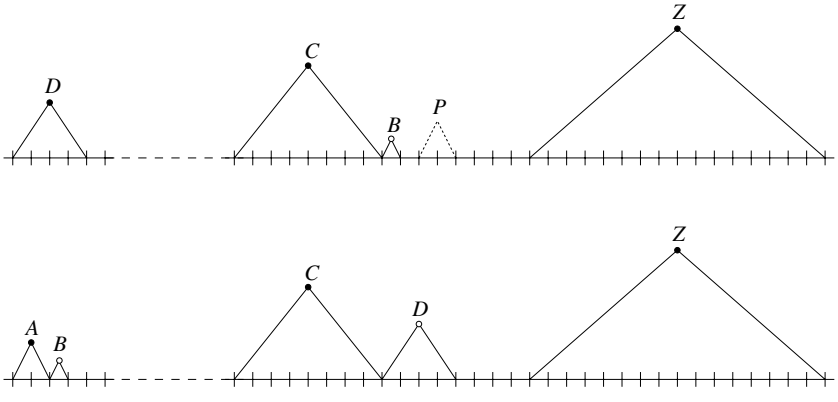


Fig. 5. An example of executing lines 33–35 of procedure INSERT

by removing A. The "pushing" is done by selecting a suitable pebble X to the right of A, removing it, and using it to fill in the gap. A new iteration then starts to fix the problem at X's original place. The suitable candidate is found as follows: from all pebbles to the right of A, the smallest one is taken; if there are more pebbles of the smallest size available, the rightmost one is chosen. The correctness is proved by showing that this procedure is well defined, and that after a finite number of iterations, a valid situation is obtained (the full proof can be found in [5]):

Theorem 2. *Consider a valid situation, and a deletion request of size a. Procedure* DELETE *correctly processes the request, resulting in a valid situation.*

Sketch of proof. Let us number the iterations of the **while** loop by $t = 1, 2, \ldots$. Let the t-th iteration starts with the configuration of pebbles and free places Γ_t, and a position i_t; it selects a pebble X_t of size x_t starting at j_t, moves it to i_t, swaps it with its white neighbor if necessary, and sets $i_{t+1} := j_t$.

Algorithm 2. Procedure DELETE removes the last pebble of level l

```
 1: let A be the last pebble of level l, and i be the starting position of A
 2: remove A
 3: while there are any pebbles to the right of i do
 4:     let x be the size of the smallest pebble to the right of i
 5:     if there is a free place of size x starting at i then
 6:         let X be the rightmost pebble of size x
 7:         let j be the starting position of X
 8:         move X to i
 9:         if X has white left neighbor Q, and Q has a left neighbor W of size x then
10:             swap X and Q                          ▷ W is black, w ≤ x
11:         end if
12:         let i:=j
13:     else
14:         exit
15:     end if
16: end while
```

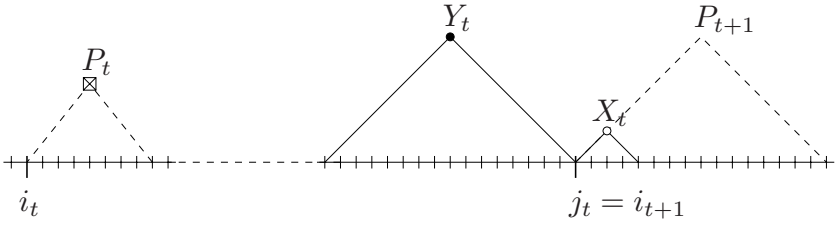


Fig. 6. One iteration of the **while** loop in procedure DELETE

Denote by P_t the free place of size a_t starting at i_t such that $a_1 = a$, and the size a_{t+1} is determined as follows: let Γ'_t be obtained from Γ_t by putting a new pebble A_t on P_t. If the color of X_t in Γ'_t is black, then $a_{t+1} := x_t$, otherwise $a_{t+1} := y_t$, where Y_t is X_t's left neighbor. It can be shown that this definition is correct, i.e. that P_{t+1} is indeed free. To do so, the following claim will be show by induction on t:

For every iteration t of the **while** *loop, the corresponding Γ'_t is a valid situation*

For $t = 1$ the iteration starts just after A was removed from a valid situation Γ and replaced by A_1 with $a_1 = a$, so the basis of induction follows.

For the inductive step, consider the t-th iteration. The algorithm either stops or enters the next iteration. We prove that in the latter case Γ'_{t+1} is valid, provided Γ'_t was valid. We distinguish two cases:

- if $x_t < a_t$, it is possible to prove that there are neither white pebbles nor free places between i_t and j_t, from which the invariants readily follow;
- if $x_t \geq a_t$, then the swap on line 10 never happens, and it is again possible to argue the validity of the resulting situation.

A separate analysis of the last iteration concludes the proof of the theorem: the algorithm stops either when there are no pebbles to the right of i_t, or when the selected pebble X_t does not fit to position i_t. In both cases it is possible to show that the resulting situation is valid. □

4 Complexity

This section is devoted to the analysis of the average number of reassignments per request needed in the worst-case computation. Obviously, INSERT requires at most constant number of reallocations. The situation with DELETE is more complicated. We develop an accounting scheme that ensures a linear (in the number of requests) number of iterations of the **while** loop over the whole computation. To this end, we introduce the notion of *coins*: each request pays a constant number of coins to the accounting system, and each individual iteration of the **while** loop in DELETE spends one coin from the system.

Each request is associated with a constant number of coins which can be put on some places. We show that it is possible to put the coins in such a way that each iteration of the loop in DELETE can be paid by an existing coin. In our coin placement strategy we shall maintain the following additional invariant:

P4: Consider a free place P such that there are some pebbles to the right of P. Let X be the smallest pebble to the right of P. Then there are at least $\lfloor 2p/x \rfloor$ coins on P.

From now on we shall consider situations with some coins placed in some places, and we show how to manage the coins so that there is always enough cash to pay for each iteration in DELETE. The following two lemmas present the accounting strategy for INSERT and DELETE:

Lemma 2. *Let us suppose that procedure* INSERT *was called from a situation in which invariants* **P1**– **P4** *hold. Then it is possible to add a constant number of coins and reallocate the existing ones in such a way that invariants* **P1**– **P4** *remain valid.*

Proof. It has already been proven that invariants **P1**– **P3** are preserved by procedure INSERT, so it is sufficient to show how to add a constant number of coins in order to satisfy **P4**. During procedure INSERT, only a constant number of pebbles are *touched* – i.e. added or reassigned. Let Γ be the situation before INSERT and Γ' be the situation after INSERT finished. Let P be a free place in Γ' and X be the smallest pebble to the right of P. We distinguish two cases:

Case 1: X **was touched during** INSERT
If $p < x/2$, no pebbles are required on P, so let us suppose that $p \geq x/2$. However, since Γ' is valid, the free bandwidth before X is less than x, so $p = x/2$, and there is only one pebble required in P; this pebble will be placed on P and charged to X. Obviously, for each touched pebble X, there may be only one free place of size $x/2$ to the left (because of **P1**), so every touched pebble will be charged at most one coin using in total constant number of coins.

Case 2: X was not touched during INSERT
If P was free in Γ the required amount of $\lfloor 2p/x \rfloor$ coins was already present on P in Γ, so let us suppose that P was not free in Γ. That means that P became free in the course of INSERT when some pebbles were reallocated. Using similar arguments as in the previous case we argue that $p = x/2$. However, during INSERT only a constant number of free places of a given size could be created, so it is affordable to put one coin on each of them. □

Lemma 3. *Let us suppose that procedure* DELETE *was called from a situation in which invariants* **P1**– **P4** *hold. Then it is possible to add a constant number of coins, remove one coin per iteration of the main loop, and reallocate the remaining coins in such a way that invariants* **P1**– **P4** *remain valid.*

Sketch of proof. Let us suppose that there is at least one full iteration of the main loop. Recall the notation from the proof of Theorem 2, i.e. we number the iterations of the main loop, and Γ_t is the configuration at the beginning of t-th iteration. Moreover, Γ'_t is obtained from Γ_t by putting a new pebble A_t on P_t. It follows from the proof of the theorem that Γ'_t is always a valid situation. We use induction on t to show that the following can be maintained:

For every iteration $t > 1$ of the **while** *loop, the corresponding Γ'_t satisfies* **P1**– **P4**, *all pebbles to the right of i_t have size at least a_t, and one extra coin lays on A_t. Moreover, if some pebble to the right of i_t has the size a_t, then two extra coins lay on A_t.*

We omit the details about the induction basis, and proceed with the inductive step. In order to prove the claim for Γ'_{t+1}, consider the situation at the end of the t-th iteration when the algorithm enters the $(t+1)$-st one. Γ'_{t+1} is obtained from Γ'_t by removing A_t, moving X_t to i_t, and placing A_{t+1} on $i_{t+1} = j_t$. Note that in this case $x_t \geq a_t$, so there is no swap. It is possible to show that all pebbles to the right of i_{t+1} have size at least a_{t+1}, and that there is no free place between i_t and j_t in Γ'_{t+1}. Since for the free places before i_t and after j_t, **P4** remains valid, we argue that **P4** holds in Γ'_{t+1}.

Now we show how to find two free coins – one to pay for the current iteration, and one to be placed on A_{t+1}. If $x_t = a_t$, there are two coins placed on A_t. Otherwise (i.e. in case that $x_t > a_t$) one coin comes from the deletion of A_t and the other can be found as follows. Since X_t was placed on i_t, and $x_t > a_t$, there must have been a free place P of size $x_t/2$ in Γ'_t. Moreover, X_t was to the right of P, and so there must have been at least one coin on P. In Γ'_{t+1}, P is covered by X_t, so the coin can be used.

The last thing to show is to find a second free coin in case that there exists some pebble Q to the right of A_{t+1} of size $q = a_{t+1}$. In this case X_t is white in Γ'_t – otherwise $a_{t+1} = x_t$ would hold and there would be no pebble of size x_t to the right of X_t. Hence $x_t \leq a_{t+1}/2$ and A_{t+1} in Γ'_{t+1} covers a place of size $a_{t+1}/2$ that has been free in Γ'_t. According to **P4** there is a coin on this place in Γ'_t; this coin can be used.

The proof of the theorem is concluded by considering the last iteration. Let $\Gamma'_{t_{fin}}$ be the last situation. The final situation is obtained from $\Gamma'_{t_{fin}}$ by removing

$A_{t_{fin}}$. We show that **P4** holds. The only free place that could violate **P4** is the one remained after $A_{t_{fin}}$, however, there was a coin on $A_{t_{fin}}$, and all pebbles to the right (if any) are bigger than $a_{t_{fin}}$. □

5 Conclusion

We have presented an online algorithm for bandwidth allocation in wireless networks, which can be used to perform the OVSF code reallocation with the amortized complexity of $O(1)$ reallocations per request. This is an improvement over the previous best known result achieving the competitive ratio of $O(n)$. Moreover, the constant in our algorithm is small enough to be of practical relevance.

On the other hand, no attempt has been made at minimizing this constant. With the best known lower bound of 1.5 it would be worthwhile to close the gap even further.

References

1. Adachi, F., Sawahashi, M., Okawa, K.: Tree structured generation of orthogonal spreading codes with different lengths for the forward link of DS-CDMA mobile radio. IEE Electronic Letters 33(1), 27–28 (1997)
2. Brodal, G.S., Demaine, E.D., Munro, J.I.: Fast allocation and deallocation with an improved buddy system. Acta Informatica 41(4–5), 273–291 (2005)
3. Defoe, D.C., Cholleti, S.R., Cytron, R.K.: Upper bound for defragmenting buddy heaps. In: LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, pp. 222–229. ACM Press, New York, USA (2005)
4. Erlebach, T., Jacob, R., Mihalák, M., Nunkesser, M., Szabó, G., Widmayer, P.: An algorithmic view on OVSF code assignment. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 270–281. Springer, Heidelberg (2004)
5. Forišek, M., Katreniak, B., Katreniaková, J., Královič, R., Královič, R., Koutný, V., Pardubská, D., Plachetka, T., Rovan, B.: Online bandwidth allocation. Technical report, arXiv:cs/0701153v1 (2007)
6. Knowlton, K.C.: A fast storage allocator. Communications of ACM 8(10), 623–624 (1965)
7. Minn, T., Siu, K.-Y.: Dynamic assignment of orthogonal variable-spreading-factor codes in W-CDMA. IEEE Journal on Selected Areas in Communications 18(8), 1429–1440 (2000)
8. Wikipedia. Universal mobile telecommunications system. Available at: From Wikipedia, the free encyclopedia [Online; accessed 23. March (2006)], http://en.wikipedia.org/wiki/Umts

Two's Company, Three's a Crowd: Stable Family and Threesome Roommates Problems

Chien-Chung Huang

Dartmouth College
villars@cs.dartmouth.edu

Abstract. We investigate Knuth's eleventh open question on stable matchings. In the stable family problem, sets of women, men, and dogs are given, all of whom state their preferences among the other two groups. The goal is to organize them into family units, so that no three of them have incentive to desert their assigned family members to join in a new family. A similar problem, called the threesome roommates problem, assumes that a group of persons, each with their preferences among the combinations of two others, are to be partitioned into triples. Similarly, the goal is to make sure that no three persons want to break up with their assigned roommates.

Ng and Hirschberg were the first to investigate these two problems. In their formulation, each participant provides a strictly-ordered list of all combinations. They proved that under this scheme, both problems are NP-complete. Their paper reviewers pointed out that their reduction exploits *inconsistent* preference lists and they wonder whether these two problems remain NP-complete if preferences are required to be consistent. We answer in the affirmative.

In order to give these two problems a broader outlook, we also consider the possibility that participants can express indifference, on the condition that the preference consistency has to be maintained. As an example, we propose a scheme in which all participants submit two (or just one in the roommates case) lists ranking the other two groups separately. The order of the combinations is decided by the sum of their ordinal numbers. Combinations are tied when the sums are equal. By introducing indifference, a hierarchy of stabilities can be defined. We prove that all stability definitions lead to NP-completeness for existence of a stable matching.

1 Problem Definition

Knuth proposed twelve open questions on the stable matching problem [9]. The eleventh question asks whether the well-studied stable marriage problem [3] can be generalized to the case of three parties, women, men, and dogs. In this paper, we call this problem the *stable family* problem and refer generically to all participants in this problem as "players." Roughly speaking, given sets of women, men, and dogs, all of whom state their preferences among the other two groups, the goal is to organize them into family units so that there is no *blocking triple*: three players each preferring one another to their assigned family members. A problem in a similar vein, which we call the *threesome roommates* problem, assumes that $3n$ students are to be assigned to the dormitory bedrooms in some college. They state their preferences of the combinations

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 558–569, 2007.

of two other persons. The goal is to partition them into sets of size 3. Such a partition (matching) is said to be stable if no three persons each prefer the others to their assigned roommates.

As Knuth does not specify any precise definition of "preference" and "blocking triples," one can conceive a number of ways to define the two problems. One possible formulation is that each player submits a strictly-ordered preference list, ranking all possible combinations that she/he/it can get in a matching. We call such a scheme strictly-ordered-complete-list (**SOCL**) scheme. In this setting, Ng and Hirschberg [10] proved that both problems are NP-complete.

At the end of their paper, Ng and Hirschberg mentioned that their reviewers pointed out their reduction allows preference to be *inconsistent*. For example, man m might rank (w_1, d_1) higher than (w_2, d_1), but he also ranks (w_2, d_2) higher than (w_1, d_2). In other words, he does not consistently prefer woman w_1 over woman w_2 (nor the other way around). Independently, Subramanian [11] gave an alternative NP-completeness proof for stable family, but his reduction also uses inconsistent lists.

The reviewers of Ng and Hirschberg wondered whether these two problems remain NP-complete if inconsistency is disallowed. To answer this open question and to motivate some variants problems we will define, we introduce the notion of *preference posets* and *simple lists*. In stable family, assuming that each player has two simple lists in which two different types of players are ranked separately, a preference poset is a product poset of the two simple lists. In such a poset, the combination (w_1, d_1) precedes another combination (w_2, d_2) only if w_1 ranks at least as high as w_2 and d_1 at least as high as d_2 in the simple lists. If neither combination precedes the other, they are incomparable. Similarly, in threesome roommates, the preference poset is the product poset of the one simple list with itself. By this notion, the question raised by the reviewers of Ng and Hirschberg can be rephrased as follows. Under the **SOCL** scheme, if every player has to submit a preference list which is a *linear extension* of her/his/its preference poset, are the stable family and the threesome roommates still NP-complete? We answer in the affirmative.

In an attempt to give these two problems a broader outlook, we then allow players to express indifference by giving full preference lists containing ties. In particular, to capture the spirit of maintaining consistency in the preferences, we stipulate that the full list must be a *relaxed linear extension* of a preference poset: strict precedence order in the poset has to be observed in the relaxed linear extension; only incomparable elements in the poset can be tied.

We propose the following scheme to make the above concept concrete. Suppose that a player submits two simple lists (or just one in the roommates case). We create a full list, ranking the combinations based on the sums of their ordinal numbers. For example, for man m, the combination of his rank-2 woman and rank-5 dog is as good as that of his rank-4 woman and rank-3 dog; while both of them are inferior to the combination of his top-ranked woman and his top-ranked dog. We call such a scheme precedence-by-ordinal-number (**PON**) scheme. The **PON** scheme produces full preference lists which are relaxed linear extensions of preference posets. Also, one can envisage an even more flexible scheme. For example, instead of giving "ranks," the players can provide "ratings" of other players. The order of the combinations can be decided by the sum of the

ratings; two combinations are tied only when the sums of their ratings are equivalent. Setting theoretical concerns aside for a moment, the above schemes are probably more practicable when n is large, because a player only has to provide lists of $\Theta(n)$ length, while under the **SOCL** scheme, they have to give strictly ordered lists of size $\Theta(n^2)$.

By allowing indifference, we can define 4 different types of blocking triples and, based on them, build up a hierarchy of stabilities. (This hierarchy is similar to that constructed by Irving in the context of 2-party stable matchings [7]).

- Weak Stable Matching: a blocking triple is one in which all three players of the blocking triple strictly prefer the other two members in the triple over their assigned family members (roommates).
- Strong Stable Matching: a blocking triple is one in which at least two players of the blocking triple strictly prefer the other two players in the triple to their assigned family members (roommates), while the remaining player can be indifferent or also strictly prefer the other two players in the triple.
- Super Stable Matching: a blocking triple is one in which at least one player of the blocking triple strictly prefers the other two players in the triple to her/his/its assigned family members (roommates), while the remaining players can be indifferent or also strictly prefer the other two players in the triple.
- Ultra Stable Matching: a blocking triple is one in which all three players in the triple are at least indifferent to the others.

Note that if ties are not allowed in the full preference lists, i.e., the **SOCL** scheme, then blocking triples can only be of degree 3. Thus there can be only one type of stability. For presentational reason, in this case, we refer to the stability under the **SOCL** scheme as the weak stability.

Our Results and Paper Roadmap. We will prove in the paper that, if full preference lists are (relaxed) linear extensions of preference posets, the problem of deciding whether weak/strong/super/ultra stable matchings exist is NP-complete in both the stable family problem and the threesome roommates problem. Our reduction techniques are inspired by Ng and Hirschberg's, although the consistency requirement in the preferences makes our construction more involved. In presenting our result, instead of directly answering the open question posed by Ng and Hirschberg's reviewers by studying weak-stability, we make a detour to first study strong/super/ultra stability. Introducing them first helps us to explain our intuition behind the more complex reduction for the former problem.

As is well-known, the stable marriage and the stable roommates problems can be solved in $O(n^2)$ time, by the Gale-Shapley algorithm [3] and by the Irving algorithm [6], respectively. Unfortunately, our results, along with Ng and Hirschberg and Subramanian's, indicate that attempts to efficiently solve the stable matching problem in generalized cases of three (or more) parties are unlikely to be fruitful. This is not surprising, as in theoretical computer science, the fine line between **P** and **NP** is often drawn between the numbers two and three.

We organize the paper as follows. In Section 2, we present necessary notation; Section 3 proves the NP-completeness of strong/super/ultra stable matchings in the stable family problem under the **PON** scheme; Section 4 presents a reduction to transform

a stable family problem to a threesome roommate problem, thus establishing the NP-completeness of strong/super/ultra stable matchings in the latter; Section 5 considers the **SOCL** scheme and proves the NP-completeness of (weak) stable matchings, thereby answering the open question posed by the anonymous reviewers of Ng and Hirschberg. Section 6 concludes and discusses related issues. Due to space constraint, we omit some proofs. See [5] for full details.

2 Preliminaries

We use $\mathcal{M}$, $\mathcal{W}$, $\mathcal{D}$ to indicate the sets of men, women, and dogs in stable family; the students in threesome roommates are denoted as $\mathcal{R}$. In stable family, $L_g(p)$ denotes the simple list of player p on the players of type $g \in \{\mathcal{M}, \mathcal{W}, \mathcal{D}\}$. For example $L_{\mathcal{W}}(m)$ is the simple list of man m among women $\mathcal{W}$. In threesome roommates, we simply write $L(m)$, where $m \in \mathcal{R}$, dropping the subscript.

In general, we use the notation $\succ$ to denote the precedence order (in either posets or in linear lists). For example, supposing that p_i ranks higher than p_j in the list l, we write $p_i \succ_l p_j$. In a poset Q, two elements q_i, q_j either one precedes the other, which we write $q_i \succ_Q q_j$ or $q_j \succ_Q q_i$, or they are incomparable, which is expressed as $q_i || _Q q_j$. The notation $\succ$ is also used to express explicitly the order of players in simple lists. For example, we write $L(p) = q \succ r \succ \cdots$ to show that player p prefers player q to player r. Note also that the notation $\cdots$ denotes the remaining players in arbitrary order. We use the notation $r_p(q)$ to indicate the rank of q on player p's simple list.

We say a blocking triple is of degree i, if i players strictly prefer the triple while the remaining $3-i$ players are indifferent. Unless stated otherwise, in the article, when we say some triple "blocks," it is always a blocking triple of degree 3.

A preference poset constructed from lists l_1 and l_2 is written as $l_1 \times l_2$. To be precise, given lists l_1 and l_2 and the poset $l_1 \times l_2$, supposing that $\{p_i, p_j\}, \{p_{i'}, p_{j'}\} \in l_1 \times l_2$, then $\{p_i, p_j\} \succ_{l_1 \times l_2} \{p_{i'}, p_{j'}\}$ only if (1) $p_i \succ_{l_1} p_{i'}, p_j = p_{j'}$, or (2) $p_j \succ_{l_2} p_{j'}, p_i = p_{i'}$, or (3) $p_i \succ_{l_1} p_{i'}, p_j \succ_{l_2} p_{j'}$. The notation $\pi(X)$ means an arbitrary permutation of elements in the set X. $E_\pi(l_1 \times l_2)$ is an arbitrary linear extension of the preference poset $l_1 \times l_2$.

3 Reducing Three-Dimensional Matching to Stable Family

In this section, we focus on the NP-completeness of strong stable matching under the **PON** scheme. Similar results hold for super stable and ultra stable matchings by a straightforward argument and will be discussed at the end of this section.

Our reduction is from the three-dimensional matching problem, one of the 21 NP-complete problems in Karp's seminal paper [8]. The problem instance is given in the form $\Upsilon = (\mathcal{M}, \mathcal{W}, \mathcal{D}, \mathcal{T})$, where $\mathcal{T} \subseteq \mathcal{M} \times \mathcal{W} \times \mathcal{D}$. The goal is to decide whether a perfect matching $\mathcal{M} \subseteq \mathcal{T}$ exists. This problem remains NP-complete even if every player in $\mathcal{M} \cup \mathcal{W} \cup \mathcal{D}$ appears exactly 2 or 3 times in the triples of $\mathcal{T}$ [4].

We first explain the intuition behind our reduction. Supposing that man m_i appears in three triples $(m_i, w_{ia}, d_{ia}), (m_i, w_{ib}, d_{ib}), (m_i, w_{ic}, d_{ic})$ in $\mathcal{T}$, we create three *dopplegangers*, m_{i1}, m_{i2}, m_{i3} in the derived stable family problem instance Υ'. We also create

four garbage collectors, $w^g_{i1}, d^g_{i1}, w^g_{i2}, d^g_{i2}$. Each doppleganger m_{ij} puts a woman-dog pair, with whom man m_i shares a triple, and the garbage collectors on top of his two simple lists. The goal of our design is that in a stable matching, exactly one doppleganger will be matched to a woman-dog pair with whom m_i shares a triple in $\mathcal{T}$, while the other two dopplegangers will be matched to garbage collectors. In the case that there are only two triples in $\mathcal{T}$ containing man m_i, we artificially make a copy of one of the triples, making the total number of triples three, and treat him as described above.

Now, we will refer to the set of dopplegangers as $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$, the set of garbage collectors as $\mathcal{W}^g_1, \mathcal{W}^g_2, \mathcal{D}^g_1, \mathcal{D}^g_2$ and the original set of real women and real dogs as $\mathcal{W}, \mathcal{D}$. Collectively, we refer to them as $X = \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3 \cup \mathcal{W}^g_1 \cup \mathcal{W}^g_2 \cup \mathcal{W} \cup \mathcal{D}^g_1 \cup \mathcal{D}^g_2 \cup \mathcal{D}$.

To realize our plan, we introduce two gadgets. The first is three sets of "dummy players": $m^\#_1, w^\#_1, d^\#_1, m^\#_2, w^\#_2, d^\#_2, m^\#_3, w^\#_3, d^\#_3$. Their preferences are such that they must be matched to one another in a stable matching. To be precise, for $j \in \{1, 2, 3\}$,

- $L_{\mathcal{W}}(m^\#_j) = w^\#_j \succ \cdots, L_{\mathcal{D}}(m^\#_j) = d^\#_j \succ \cdots$
- $L_{\mathcal{M}}(w^\#_j) = m^\#_j \succ \cdots, L_{\mathcal{D}}(w^\#_j) = d^\#_j \succ \cdots$
- $L_{\mathcal{M}}(d^\#_j) = m^\#_j \succ \cdots, L_{\mathcal{W}}(d^\#_j) = w^\#_j \succ \cdots$

These nine dummy players are used to "pad" the preference lists of other players. Their purpose will be clear shortly.

Another gadget we need is a set of "guard players" for each doppleganger in $\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3$. They will make sure that in a stable matching, a doppleganger m_{ij} will only get a woman-dog pair with whom m_i shares a triple in $\mathcal{T}$ or those garbage collectors. As an example, consider the doppleganger m_{i1}. He has six associated guard players, $m^{\flat 1}_{i1}, w^{\flat 1}_{i1}, d^{\flat 1}_{i1}, m^{\flat 2}_{i1}, w^{\flat 2}_{i1}, d^{\flat 2}_{i1}$ and their preferences are summarized below:

- $L_{\mathcal{W}}(m_{i1}) = w^g_{i2} \succ w^g_{i1} \succ w_{ia} \succ w^{\flat 1}_{i1} \succ w^{\flat 2}_{i1} \succ w^\#_1 \succ w^\#_2 \succ w^\#_3 \succ \cdots,$
  $L_{\mathcal{D}}(m_{i1}) = d^g_{i2} \succ d^g_{i1} \succ d_{ia} \succ d^{\flat 2}_{i1} \succ d^{\flat 1}_{i1} \succ d^\#_1 \succ d^\#_2 \succ d^\#_3 \succ \cdots$
- $L_{\mathcal{W}}(m^{\flat 1}_{i1}) = w^{\flat 1}_{i1} \succ \cdots, L_{\mathcal{D}}(m^{\flat 1}_{i1}) = d^{\flat 1}_{i1} \succ \cdots$
  $L_{\mathcal{W}}(m^{\flat 2}_{i1}) = w^{\flat 2}_{i1} \succ \cdots, L_{\mathcal{D}}(m^{\flat 2}_{i1}) = d^{\flat 2}_{i1} \succ \cdots$
- $L_{\mathcal{M}}(w^{\flat 1}_{i1}) = m_{i1} \succ m^{\flat 1}_{i1} \succ \cdots, L_{\mathcal{D}}(w^{\flat 1}_{i1}) = d^{\flat 1}_{i1} \succ d^\#_1 \succ \cdots$
  $L_{\mathcal{M}}(w^{\flat 2}_{i1}) = m_{i1} \succ m^{\flat 2}_{i1} \succ \cdots, L_{\mathcal{D}}(w^{\flat 2}_{i1}) = d^{\flat 2}_{i1} \succ d^\#_1 \succ \cdots$
- $L_{\mathcal{M}}(d^{\flat 1}_{i1}) = m_{i1} \succ m^{\flat 1}_{i1} \succ \cdots, L_{\mathcal{W}}(d^{\flat 1}_{i1}) = w^{\flat 1}_{i1} \succ w^\#_1 \succ \cdots$
  $L_{\mathcal{M}}(d^{\flat 2}_{i1}) = m_{i1} \succ m^{\flat 2}_{i1} \succ \cdots, L_{\mathcal{W}}(d^{\flat 2}_{i1}) = w^{\flat 2}_{i1} \succ w^\#_1 \succ \cdots$

The following case analysis proves that, in a stable matching M', m_{i1} will get only players from the set $\{w^g_{i1}, w^g_{i2}, w_{ia}, d^g_{i1}, d^g_{i2}, d_{ia}\}$.

- Suppose that m_{i1} gets two players ranking below $w^{\flat 1}_{i1}$ and $d^{\flat 1}_{i1}$ respectively. It can be observed that for both $w^{\flat 1}_{i1}, d^{\flat 1}_{i1}$, the best man is m_{i1}. Therefore, they would prefer m_{i1} and so does he them, inducing a blocking triple to M', a contradiction.
- Suppose that m_{i1} gets a woman $w \in \{w_{ia}, w^g_{i1}, w^g_{i2}\}$ and a dog d ranking below $d^{\flat 1}_{i1}$. In this case, we can be sure that d cannot be $d^\#_1$ or $d^\#_2$ or $d^\#_3$, since their preferences guarantee that they will only be matched to other dummy players. So, $r_{m_{i1}}(w) + r_{m_{i1}}(d) \geq 10$, while $r_{m_{i1}}(w^{\flat 1}_{i1}) + r_{m_{i1}}(d^{\flat 1}_{i1}) = 9$, causing $(m_{i1}, w^{\flat 1}_{i1}, d^{\flat 1}_{i1})$

to become a blocking triple. This example explains why we need to pad the simple lists of m_{i1} with dummy players.
The case that m_{i1} gets a dog $d \in \{d_{ia}, d^g_{i1}, d^g_{i2}\}$ and a woman w ranking lower than $w^{\flat 2}_{i1}$ follows analogous arguments; $(m_{i1}, w^g_{i2}, d^g_{i2})$ will become a blocking triple.

– Suppose that m_{i1} gets only one of the players from the set $\{w^{\flat 1}_{i1}, w^{\flat 2}_{i1}, d^{\flat 1}_{i1}, d^{\flat 2}_{i1}\}$. Without loss of generality, we assume that $(m_{i1}, w^{\flat 1}_{i1}, d^{\phi}), d^{\phi} \neq d^{\flat 1}_{i1}$, is part of the matching. For woman $w^{\flat 1}_{i1}$, dog d^{ϕ} cannot be the dummy player $d^{\#}_1$. Therefore, $r_{w^{\flat 1}_{i1}}(m_{i1}) + r_{w^{\flat 1}_{i1}}(d^{\phi}) \geq 4 > 3 = r_{w^{\flat 1}_{i1}}(m^{\flat 1}_{i1}) + r_{w^{\flat 1}_{i1}}(d^{\flat 1}_{i1})$. Similarly for $d^{\flat 1}_{i1}$, $r_{d^{\flat 1}_{i1}}(m^{\flat 1}_{i1}) + r_{d^{\flat 1}_{i1}}(w^{\flat 1}_{i1}) = 3$, which is better than whatever combination it can get. Therefore, we have that $(m^{\flat 1}_{i1}, w^{\flat 1}_{i1}, d^{\flat 1}_{i1})$ constitutes a blocking triple to M'. This example shows why we need to pad the preference of $w^{\flat 1}_{i1}, d^{\flat 1}_{i1}$ (and also $w^{\flat 2}_{i1}, d^{\flat 2}_{i1}$) with dummy players.
– Suppose that m_{i1} gets $w^{\flat 1}_{i1}$ and $d^{\flat 1}_{i1}$. Note that $w^{\flat 1}_{i1} \succ w^{\flat 2}_{i1}$ and $d^{\flat 2}_{i1} \succ d^{\flat 1}_{i1}$. Therefore, m_{i1} is *indifferent* to the combinations of $w^{\flat 2}_{i1}$ and $d^{\flat 2}_{i1}$, since $r_{m_{i1}}(w^{\flat 1}_{i1}) + r_{m_{i1}}(d^{\flat 1}_{i1}) = 9 = r_{m_{i1}}(w^{\flat 2}_{i1}) + r_{m_{i1}}(d^{\flat 2}_{i1})$. Additionally, $w^{\flat 2}_{i1}, d^{\flat 2}_{i1}$ strictly prefer m_{i1}. Hence $(m_{i1}, w^{\flat 2}_{i1}, d^{\flat 2}_{i1})$ constitutes a blocking triple of degree 2 to M'. This explains why we need two sets of guard players to guarantee that the doppleganger will "behave" in a stable matching.
The case that m_{i1} gets $w^{\flat 2}_{i1}$ and $d^{\flat 2}_{i1}$ follows analogous arguments.

The other two dopplegangers m_{i2}, m_{i3} also have six associated guard players for each; they, along with their associated guard players, have similar preferences to guarantee that m_{i2} and m_{i3} will only get garbage collectors or the woman-dog pairs with whom m_i shares triples. The only difference in the lists is that m_{i2} and m_{i3} replace w_{ia}, d_{ia} with w_{ib}, d_{ib}, and with w_{ic}, d_{ic}, respectively, in their simple lists. For a summary of the simple lists of members in the set X, see Table 1. It should be noted that w^g_{i1}, d^g_{i1} (and also w^g_{i2}, d^g_{i2}) rank the three dopplegangers in reverse order. This trick guarantees that the dopplegangers will not form blocking triples with the garbage collectors, defeating our purpose. For example, suppose (m_{i1}, w_{ia}, d_{ia}) is part of the matching, we want to avoid $(m_{i1}, w^g_{i1}, d^g_{i1})$ to becoming a blocking triple. It can be easily verified that if w^g_{i1} and d^g_{i1} are matched to m_{i2} or m_{i3}, such a blocking triple will not be formed.

Finally, garbage collectors also use dummy players to pad their simple lists, to avoid the awkward situation that some doppleganger is matched to a real woman and a garbage collector dog (or a real dog and a garbage collector woman). How this arrangement works will be clear in the proof below.

Lemma 1. *Suppose a stable matching M' exists in the derived stable family problem instance Υ'. The following facts hold in M':*

– *Fact A: The three sets of dummy players are matched to one another.*
– *Fact B: For each doppleganger $m_{ij} \in \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3$, the ranks of his family members in M' are at least as high as 3 in his simple lists.*
– *Fact C: The six associated guard players of each doppleganger $m_{ij} \in \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3$ are matched to one another.*

Table 1. The simple lists of all players in the set $X = \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3 \cup \mathcal{W}_1^g \cup \mathcal{W}_2^g \cup \mathcal{W} \cup \mathcal{D}_1^g \cup \mathcal{D}_2^g \cup \mathcal{D}$. We assume that there exist three triples $(m_i, w_{ia}, d_{ia}), (m_i, w_{ib}, d_{ib}), (m_i, w_{ic}, d_{ic})$ in $\mathcal{T}$.

Player	Simple Lists
$m_{i1} \in \mathcal{M}_1$	$L_{\mathcal{W}}(m_{i1}) {=} w_{i2}^g \succ w_{i1}^g \succ w_{ia} \succ w_{i1}^{\flat 1} \succ w_{i1}^{\flat 2} \succ w_1^{\#} \succ w_2^{\#} \succ w_3^{\#} \succ \cdots$ $L_{\mathcal{D}}(m_{i1}) = d_{i2}^g \succ d_{i1}^g \succ d_{ia} \succ d_{i1}^{\flat 2} \succ d_{i1}^{\flat 1} \succ d_1^{\#} \succ d_2^{\#} \succ d_3^{\#} \succ \cdots$
$m_{i2} \in \mathcal{M}_1$	$L_{\mathcal{W}}(m_{i2}) {=} w_{i2}^g \succ w_{i1}^g \succ w_{ib} \succ w_{i2}^{\flat 1} \succ w_{i2}^{\flat 2} \succ w_1^{\#} \succ w_2^{\#} \succ w_3^{\#} \succ \cdots$ $L_{\mathcal{D}}(m_{i2}) = d_{i2}^g \succ d_{i1}^g \succ d_{ib} \succ d_{i2}^{\flat 2} \succ d_{i2}^{\flat 1} \succ d_1^{\#} \succ d_2^{\#} \succ d_3^{\#} \succ \cdots$
$m_{i3} \in \mathcal{M}_1$	$L_{\mathcal{W}}(m_{i3}) {=} w_{i2}^g \succ w_{i1}^g \succ w_{ic} \succ w_{i3}^{\flat 1} \succ w_{i3}^{\flat 2} \succ w_1^{\#} \succ w_2^{\#} \succ w_3^{\#} \succ \cdots$ $L_{\mathcal{D}}(m_{i3}) = d_{i2}^g \succ d_{i1}^g \succ d_{ic} \succ d_{i3}^{\flat 2} \succ d_{i3}^{\flat 1} \succ d_1^{\#} \succ d_2^{\#} \succ d_3^{\#} \succ \cdots$
$w_{i1}^g \in \mathcal{W}_1^g$	$L_{\mathcal{M}}(w_{i1}^g) = m_{i1} \succ m_{i2} \succ m_{i3} \succ \cdots$ $L_{\mathcal{D}}(w_{i1}^g) = d_{i1}^g \succ d_1^{\#} \succ d_2^{\#} \succ d_3^{\#} \succ \cdots$
$d_{i1}^g \in \mathcal{D}_1^g$	$L_{\mathcal{M}}(d_{i1}^g) = m_{i3} \succ m_{i2} \succ m_{i1} \succ \cdots$ $L_{\mathcal{W}}(d_{i1}^g) = w_{i1}^g \succ w_1^{\#} \succ w_2^{\#} \succ w_3^{\#} \succ \cdots$
$w_{i2}^g \in \mathcal{W}_2^g$	$L_{\mathcal{M}}(w_{i2}^g) = m_{i1} \succ m_{i2} \succ m_{i3} \succ \cdots$ $L_{\mathcal{D}}(w_{i2}^g) = d_{i2}^g \succ d_1^{\#} \succ d_2^{\#} \succ d_3^{\#} \succ \cdots$
$d_{i2}^g \in \mathcal{D}_2^g$	$L_{\mathcal{M}}(d_{i2}^g) = m_{i3} \succ m_{i2} \succ m_{i1} \succ \cdots$ $L_{\mathcal{W}}(d_{i2}^g) = w_{i2}^g \succ w_1^{\#} \succ w_2^{\#} \succ w_3^{\#} \succ \cdots$
$w \in \mathcal{W}$	$L_{\mathcal{M}}(w) = \cdots$ $L_{\mathcal{D}}(w) = \cdots$
$d \in \mathcal{D}$	$L_{\mathcal{M}}(d) = \cdots$ $L_{\mathcal{W}}(d) = \cdots$

Proof. Fact A follows directly from construction. Fact B is true as we have argued in the case analysis before. Fact C is true because if the guard players are not matched to one another, they will block M', unless $w_{ij}^{\flat 1}, d_{ij}^{\flat 1}$ or $w_{ij}^{\flat 2}, d_{ij}^{\flat 2}$ are matched to m_{ij} in M', but this is impossible because of Fact B. □

Lemma 2. *Suppose a stable matching M' exists in the derived stable family problem instance Υ'. Consider the garbage collectors $w_{i1}^g, d_{i1}^g, w_{i2}^g, d_{i2}^g$ created for man $m_i \in \mathcal{M}$. We must have that w_{i1}^g, d_{i1}^g belong to the same triple t_1 and that w_{i2}^g, d_{i2}^g belong to the same triple t_2 in M'. Moreover, in t_1 and t_2, the man player must be one of the dopplegangers m_{i1}, m_{i2} and m_{i3}.*

Proof. We will prove this lemma by progressively establishing the following facts.

Fact D: *w_{i2}^g and d_{i2}^g must belong to the same triple t_2 in M'.*

Proof: For a contradiction, suppose that w_{i2}^g and d_{i2}^g are in different triples in M'. We claim that $(m_{i1}, w_{i2}^g, d_{i2}^g)$ forms a blocking triple. It is obvious that m_{i1} and w_{i2}^g prefer such a triple. Now let the man and woman partners of d_{i2}^g be m^ϕ and $w^\phi \neq w_{i2}^g$; then by Fact A in Lemma 1, $r_{d_{i2}^g}(w) \geq 5$. We have that $r_{d_{i2}^g}(m_{i1}) + r_{d_{i2}^g}(w_{i2}^g) = 4 < 6 \leq r_{d_{i2}^g}(m^\phi) + r_{d_{i2}^g}(w^\phi)$. So d_{i2}^g will also prefer m_{i1} and w_{i2}^g, forming a blocking triple with them to M'. This proof also shows why we need to pad the preferences of the garbage collectors.

Fact E: *w^g_{i1} and d^g_{i1} must belong to the same triple t_1 in M'.*

Proof: For a contradiction, suppose that $(m^{\phi 1}, w^g_{i1}, d^{\phi 1})$ and $(m^{\phi 2}, w^{\phi 2}, d^g_{i1})$ are triples in M'. There exists at least one doppleganger in $\{m_{i1}, m_{i2}, m_{i3}\}$ preferring the combination of w^g_{i1} and d^g_{i1} (since at most one doppleganger can be matched to w^g_{i2} and d^g_{i2}). Let such a doppleganger be m_{ij}. Then by Fact A in Lemma 1, $r_{w^g_{i1}}(m_{ij}) + r_{w^g_{i1}}(d^g_{i1}) \leq 4 < 6 \leq r_{w^g_{i1}}(m^{\phi 1}) + r_{w^g_{i1}}(d^{\phi 2})$; and similarly, $r_{d^g_{i1}}(m_{ij}) + r_{d^g_{i1}}(w^g_{i1}) \leq 4 < 6 \leq r_{d^g_{i1}}(m^{\phi 2}) + r_{d^g_{i1}}(w^{\phi 2})$, implying that $(m_{ij}, w^g_{i1}, d^g_{i1})$ blocks M'.

Fact F: *w^g_{i2} and d^g_{i2} must be matched to one of the dopplegangers of m_i in M', and so are w^g_{i1} and d^g_{i1}.*

Proof: If w^g_{i2} and d^g_{i2} are not matched to a doppleganger of m_i, then any doppleganger m_{ij} will prefer the combination of them over his family members, causing $(m_{ij}, w^g_{i2}, d^g_{i2})$ to block M'. A similar argument applies to the case of w^g_{i1} and d^g_{i1}, giving the lemma. □

By the previous two lemmas, we have established the correctness of the reduction on one side.

Lemma 3. *(Sufficiency) If there exists a stable matching M' in the derived stable family problem instance Υ', there exists a perfect matching M in the original three-dimensional matching instance Υ.*

To show the necessity, we need to prove one more lemma.

Lemma 4. *In a matching M' in the derived stable family problem instance Υ', suppose that dummy players are matched to one another. Suppose further that the garbage collectors of m_i are matched to two of the dopplegangers of m_i, while the remaining doppleganger m_{ij} is matched to a real woman and a real dog with whom m_i shares a triple in $\mathcal{T}$ in the original three-dimensional matching instance Υ. Then there is no blocking triple in which the dopplegangers m_{i1}, m_{i2}, and m_{i3} are involved.*

Proof. We assume that $(m_{i1}, w^g_{i2}, d^g_{i2}), (m_{i2}, w^g_{i1}, d^g_{i1}), (m_{i3}, w_{ic}, d_{ic}) \in M'$. Other cases follow analogous arguments. We claim that there does not exist a blocking triple of the form $(m_{ij}, w^g_{i1}, d^{\phi 1})$, $(m_{ij}, w^g_{i2}, d^{\phi 2})$, $(m_{ij}, w^{\phi 3}, d^g_{i1})$, and $(m_{ij}, w^{\phi 4}, d^g_{i2})$ where $d^{\phi 1} \neq d^g_{i1}$, $d^{\phi 2} \neq d^g_{i2}$, $w^{\phi 3} \neq w^g_{i1}$, and $w^{\phi 4} \neq w^g_{i2}$. We only argue the first case. Since $d^{\phi 1} \notin \{d^\#_1, d^\#_2, d^\#_3\}$, we have $r_{w^g_{i1}}(d^{\phi 1}) \geq 5 > 3 = r_{w^g_{i1}}(d^g_{i1}) + r_{w^g_{i1}}(m_{i2})$. Therefore, w^g_{i1} has no incentive to join the combination of m_{ij} and $d^{\phi 1}$.

Now we only need to consider the three remaining potential blocking triples: $(m_{i2}, w^g_{i2}, d^g_{i2}), (m_{i3}, w^g_{i2}, d^g_{i2}), (m_{i3}, w^g_{i1}, d^g_{i1})$. It can be easily verified that they do not block M' because the orders of the three dopplegangers in the simple lists of w^g_{i1} and d^g_{i1} (and also w^g_{i2} and d^g_{i2}) are reversed. □

Lemma 5. *(Necessity) Suppose that there is a perfect matching M in the original three-dimensional matching instance Υ. There also exists a stable matching M' in the derived stable family problem instance Υ'.*

Proof. We build a stable matching M' in Υ' as follows. Let the dummy players $\{m_j^{\#}, w_j^{\#}, d_j^{\#}\}, 1 \leq j \leq 3$, be matched to one another. Given any doppleganger m_{ij}, let his guard players $\{m_{ij}^{\flat 1}, w_{ij}^{\flat 1}, d_{ij}^{\flat 1}\}, \{m_{ij}^{\flat 2}, w_{ij}^{\flat 2}, d_{ij}^{\flat 2}\}$ be matched to one another as well. Furthermore, suppose that $(m_i, w_{ix}, d_{ix}) \in M$. Let the doppleganger who lists w_{ix} and d_{ix} above his guard players be matched to w_{ix} and d_{ix}, while the other two dopplegangers be matched to the garbage collectors. By this construction, it can be seen that none of the guard players and dummy players will be part of a blocking triple. This, combined with Lemma 4, completes the proof. □

Suppose that in the given three-dimensional matching instance Υ, $|\mathcal{M}| = |\mathcal{W}| = |\mathcal{D}| = n$. Then in the derived instance Υ', we use in all $3n$ dopplegangers, $18n$ guard players, $4n$ garbage collectors, $2n$ real women and real dogs, and 9 dummy players. Their preferences (in the form of simple lists) can be generated in $O(n^2)$ time. Therefore, this is a polynomial-time reduction. Also, given any matching, we definitely can check its stability in $O(n^3)$ time. Combining the two facts with Lemma 3 and Lemma 5, we can conclude:

Theorem 1. *It is NP-complete to decide whether strong stable matchings exist under the **PON** scheme. Therefore, the question of deciding existence of strong stable matching is also NP-complete when the full preference lists are consistent, i.e., when they are relaxed linear extensions of preference posets.*

Super Stability and Ultra Stability. It can be observed that throughout the proof, all arguments involving blocking triples use those of degree 3. The only exception is the occasion that we argue that a doppleganger cannot be matched to his guard players in a stable matching. To recall, supposing that $(m_{ij}, w_{ij}^{\flat 1}, d_{ij}^{\flat 1})$ is part of a matching, then $(m_{ij}, w_{ij}^{\flat 2}, d_{ij}^{\flat 2})$ is a blocking triple of degree 2. (Or if the latter is part of the matching, the former is a blocking triple of degree 2). Therefore, our reduction only uses blocking triples of degree 2 or 3; both are still blocking triples with regard to super stability and ultra stability. Moreover, when we argue the strong-stability of matchings in the reduction, we never allow blocking triples of degree 0 or degree 1 to exist. Therefore, essentially, our reduction has also established the NP-completeness of super stable matchings and ultra stable matchings.

4 Threesome Roommates with Relaxed Linear Extensions of Preference Posets

In this section, we exhibit a reduction of stable family to threesome roommates, thereby establishing the NP-completeness of strong/super/ultra stable matchings in the latter problem. Instead of the **PON** scheme, we use the more general scheme in which any relaxed linear extension of preference posets is allowed. We choose to use this scheme because the involved reduction technique has a different flavor. Nonetheless, we do have another reduction for the **PON** scheme. See [5] for details.

Let an instance of stable family problem be $\Upsilon = (\mathcal{M}, \mathcal{W}, \mathcal{D}, \Psi)$, where Ψ represents the preferences of the players in $\mathcal{M} \cup \mathcal{W} \cup \mathcal{D}$. We create an instance of threesome

roommates $\Upsilon' = (\mathcal{R}', \Psi')$ by copying all players in $\mathcal{M} \cup \mathcal{W} \cup \mathcal{D}$ into $\mathcal{R}'$. Regarding the preferences in Ψ', we first build up the simple lists of all players.

- Suppose $m \in \mathcal{M}, L(m) = L_{\mathcal{W}}(m) \succ L_{\mathcal{D}}(m) \succ \pi(\mathcal{M} - \{m\})$.
- Suppose $w \in \mathcal{W}, L(w) = L_{\mathcal{D}}(w) \succ L_{\mathcal{M}}(w) \succ \pi(\mathcal{W} - \{w\})$.
- Suppose $d \in \mathcal{D}, L(d) = L_{\mathcal{M}}(d) \succ L_{\mathcal{W}}(d) \succ \pi(\mathcal{D} - \{d\})$.

In words, a man lists all women and then all dogs, based respectively on their original order in his simple lists in Ψ. He then attaches other fellow men in arbitrary order to the end of his list. Women and dogs have analogous arrangements in their simple lists.

Having constructed the simple lists, we still need to build consistent relaxed linear extensions. We will exploit the following lemma, whose proof can be found in the full version [5].

Lemma 6. *Let l be a strictly-ordered list. Suppose that l is decomposed into nonempty contiguous sublists $(l_1, l_2, \cdots, l_k)$ such that (1) $\bigcup_{i=1}^{k} l_i = l$, (2) if $e \succ_{l_i} f$, then $e \succ_l f$, and (3) if $e \in l_i, f \in l_j, i < j$, then $e \succ_l f$. Then there exists a linear extension of $l \times l$ such that all combinations drawn from $\{l_i, l_j\}$ precede all pairs drawn from $\{l_{i'}, l_{j'}\}$, provided that $i \leq j, i' \leq j'$ and one of the following conditions holds (1) $i < i'$, (2) $i = i', j < j'$.*

By Lemma 6, we can construct the linear extensions as follows:

- Consider $m \in \mathcal{M}$ and assume that $W = L_{\mathcal{W}}(m), D = L_{\mathcal{D}}(m), N = \pi(\mathcal{M} - \{m\})$. His relaxed linear extension is: $E_\pi(W \times W) \succ X \succ E_\pi(W \times N) \succ E_\pi(D \times D) \succ E_\pi(D \times N) \succ E_\pi(N \times N)$, where X is the original relaxed linear extension of man m's preference poset given in Ψ.
- Consider $w \in \mathcal{W}$ and assume that $D = L_{\mathcal{D}}(w), N = L_{\mathcal{M}}(w), W = \pi(\mathcal{W} - \{w\})$. Her relaxed linear extension is: $E_\pi(D \times D) \succ Y \succ E_\pi(D \times W) \succ E_\pi(N \times N) \succ E_\pi(N \times W) \succ E_\pi(W \times W)$, where Y is the original relaxed linear extension of woman w's preference poset given in Ψ.
- Consider $d \in \mathcal{D}$ and assume that $N = L_{\mathcal{M}}(d), W = L_{\mathcal{W}}(d), D = \pi(\mathcal{D} - \{d\})$. Its relaxed linear extension is: $E_\pi(N \times N) \succ Z \succ E_\pi(N \times D) \succ E_\pi(W \times W) \succ E_\pi(W \times D) \succ E_\pi(D \times D)$, where Z is the original relaxed linear extension of dog d's preference poset given in Ψ.

To prove that the reduction from Υ to Υ' is valid, we will rely heavily on the following technical lemma.

Lemma 7. *In the derived instance Υ', if a stable matching M' exists, every triple in M' must contain a man, a woman, and a dog. Moreover, suppose that in a matching M'' in Υ' in which each player gets two other types of players as roommates, then a blocking triple cannot contain two (or three) players of the same type.*

Proof. For the first part, we argue case by case.

1. If $\{m, w_i, w_j\} \in M'$, there exists another man m' who can get neither a woman-woman combination nor a woman-dog combination. By construction, m' would

prefer any woman-dog combination to his assigned roommates in M'. Similarly, there exists a dog d' who gets another fellow dog in M'. Such a dog would prefer a man-woman combination to its assigned roommates in M'. Finally, woman w_i and w_j would prefer a dog-man combination. Therefore, both $\{m', w_i, d'\}$ and $\{m', w_j, d'\}$ block M', a contradiction.
2. If $\{m, m_i, m_j\} \in M'$, then there exists a woman w who gets a fellow woman in M' and a dog d who gets a fellow dog in M'. Thus, woman w would prefer a dog-man combination and dog d would prefer a man-woman combination. Therefore, $\{m, w, d\}, \{m_i, w, d\}, \{m_j, w, d\}$ block M', a contradiction.
3. All other cases can be argued similarly.

For the second part, suppose that matching M'' has the stated property. Given any man m, by our construction, if there is a blocking triple containing m and in which there are two players of the same type, the only possibility of a blocking triple is $\{m, w_i, w_j\}$. However, neither w_i nor w_j would prefer such a triple, because in our construction, for a woman, a dog-man combination is better than a man-woman combination. The other potential blocking triples not involving men follow analogous arguments, thus giving us the lemma. □

It is straightforward to use Lemma 7 to prove our reduction is a valid one.

Theorem 2. *Deciding whether strong/super/ultra stable matchings exist in the threesome roommates problem is NP-complete when full preference lists are consistent, i.e., when they are relaxed linear extension of preference posets.*

5 Weak Stability Under the SOCL Scheme

Due to space constraint, we can only state our results and leave the details to the full version [5].

Theorem 3. *It is NP-complete to decide whether weak stable matchings exist under the **SOCL** scheme, for both the stable family and the threesome roommates problems. Hence, it is also NP-complete to decide whether a weak stable matching exists when consistent preferences are allowed to contain ties: i.e. the full preferences are relaxed linear extensions of preference posets.*

6 Conclusion and Related Problems

In this paper, we answer the open question of whether the stable family and the threesome roommates problems are NP-complete if all players have to provide consistent preference lists. We introduce a scheme in which players can express indifference on the precondition that their preferences have to be consistent. Under this scheme, a variety of stabilities are defined and we prove that all lead to NP-complete problems.

Since we have proved that the general cases of stable family and threesome roommates are NP-complete, a natural question to ask is whether there are special cases that allow polynomial time solutions. Actually, examples of the two problems that can be solved efficiently do exist.

Consider the following scheme. Every player submits two simple lists. A man evaluates combinations first by the woman he gets, then by the dog; a woman first by the man she gets, then by the dog; a dog first by the man it gets, then by the woman. (Note the asymmetry). It is not hard to see that we can apply the Gale-Shapley algorithm twice to get a weak stable matching: letting the men propose to women and then propose to dogs. Women and dogs make the decision of acceptance or rejection based on their simple lists of men [2]. Merging the two matchings will give a stable matching in the stable family problem.

However, even a little twist can make the above scheme hard to solve. Suppose a man decides first based on the woman he gets and then the dog; a woman first based on the dog she gets and then on the man; a dog decides first based on the man it gets then on the woman. The Gale-Shapley algorithm no longer works [1].

Interestingly, the above scheme is reminiscent of another open problem allegedly originated by Knuth. Suppose that a man has only a simple list for women; a woman has only a simple list for dogs; a dog has only a simple list for men. This problem is called *circular stable matching*. Its complexity is still unknown.

Acknowledgements

I thank my adviser Peter Winkler for many helpful suggestions. I am also indebted to the anonymous reviewers for their detailed comments.

References

1. Boros, E., Gurvich, V., Jaslar, S., Krasner, D.: Stable matchings in three-sided systems with cyclic preferences. Discrete Mathematics 289(1-3), 1–10 (2004)
2. Danilov, V.I.: Existence of stable matchings in some three-sided systems. Mathematical Social Science 46(2), 145–148 (2003)
3. Gale, D., Shapley, L.: College admissions and the stability of marriage. American Mathematical Monthly 69(1), 9–15 (1962)
4. Garey, M., Johnson, D.: Computers and Intractablility. Freeman, San Francisco (1979)
5. Huang, C.-C.: Two's company, three's a crowd: Stable family and threesome roommates problems. Technical Report TR2007-598, Computer Science Department, Dartmouth College (2007)
6. Irving, R.: An efficient algorithm for the stable room-mates problem. Journal of Algorithms 6, 577–595 (1985)
7. Irving, R.: Stable marriage and indifference. Discrete Applied Mathematics 48, 261–272 (1994)
8. Karp, R.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, pp. 85–103 (1972)
9. Knuth, D.: Mariages stables et leurs relations avec d'autre problèmes combinatoires. Les Presses de l'université de Montréal (1976)
10. Ng, C., Hirschberg, D.: Three-dimensional stable matching problems. SIAM Journal on Discrete Mathematics 4(2), 245–252 (1991)
11. Subramanian, A.: A new approach to stable matching problems. SIAM Journal on Computing 23(4), 671–700 (1994)

On the Complexity of Sequential Rectangle Placement in IEEE 802.16/WiMAX Systems

Amos Israeli[1], Dror Rawitz[2,⋆], and Oran Sharon[1]

[1] Department of Computer Science, Netanya Academic College, Netanya 42100, Israel
{amos,oran}@netanya.ac.il
[2] School of Electrical Engineering, Tel Aviv University, Tel Aviv 69978, Israel
rawitz@eng.tau.ac.il

Abstract. We study the problem of scheduling transmissions on the Downlink of IEEE 802.16/WiMAX systems that use the OFDMA technology. These transmissions are scheduled using a matrix whose dimensions are frequency and time, where every matrix cell is a time slot on some carrier channel. The IEEE 802.16 standard mandates that (i) every transmission occupies a rectangular set of cells, and (ii) transmissions must be scheduled according to a given order. We show that if the number of cells required by a transmission is not limited (up to the matrix size), the problem of maximizing matrix utilization is very hard to approximate. On the positive side we show that if the number of cells of every transmission is limited to some constant fraction of the matrix area, the problem can be approximated to within a constant factor. As far as we know this is the first paper that considers this *sequential rectangle placement problem*.

1 Introduction

Background and motivation. The IEEE 802.16/WiMAX system [1] is an emerging standard for Wireless Local Loop (WLL) systems [2] that are designed to enable residential and business subscribers Broadband Wireless Access (BWA) to core networks, e.g., the public telephone network and the Internet. An IEEE 802.16 system consists of a Base Station (BS) and Subscriber Systems (SSs) as depicted in Fig. 1. The wireless link from the BS to the SSs (from the SSs to the BS, respectively) is called the Downlink (Uplink, respectively). One of the options to realize the physical layer in 802.16 is Orthogonal Frequency Division Multiplexing Access (OFDMA), which is a form of multicarrier modulation. In OFDMA, the transmission bandwidth on the Downlink is divided into several subchannels that are used by the BS to transmit to the SSs in parallel. The transmission time over each subchannel is further divided into time slots. A predefined number of slots from all the subchannels together are grouped into periods called *Subframes*. The same holds respectively on the Uplink.

⋆ Part of the work on this paper was done while the author was a postdoc at the Caesarea Rothschild Institute, University of Haifa.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 570–581, 2007.

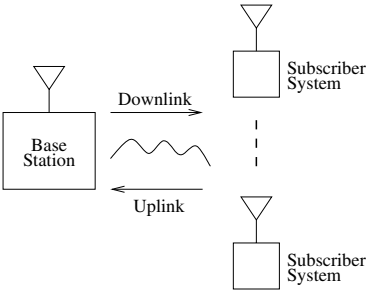


Fig. 1. The physical architecture of an 802.16 system

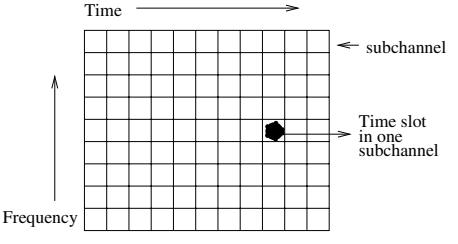


Fig. 2. The Downlink time/frequency matrix in OFDMA

Notice that the Downlink subframe is actually a time/frequency matrix, which for the sake of brevity is called from now on simply the *matrix*. The matrix' frequency dimension is equal to the number of the OFDMA subchannels, while the time dimension is equal to the number of time slots in each Downlink subframe. An example of a matrix is given in Fig. 2.

In each time slot the BS may transmit some fixed number of bits in some subchannel. We refer to such a time slot (in some subchannel) as a *cell*. Each individual transmission of the BS to some SS is called a *packet*. In order to transmit packets on the Downlink, each packet should be assigned a set of matrix cells on which the packet will be transmitted, and all sets must be disjoint. Every SS listens for some predefined number of time slots on predefined subchannels in order to receive packets destined to itself. These time slots and subchannels can change from one Downlink subframe to another.

The assignment packets to sets of matrix cells should follow some requirements: for each packet transmitted on the Downlink, the IEEE 802.16 standard mandates that the set of scheduling matrix cells allocated for the packet transmission must be rectangular. Further, an IEEE 802.16 system is intended to support various "high level" protocols, such as ATM and IP. Thus, the system is supposed to have QoS capabilities for, e.g., ATM VCs. This means that the BS should be able to transmit packets according to some scheduling disciplines that guarantee delay bounds, such as Delay-Earliest-Due-Date [3] in which every packet is assigned a deadline for its transmission, and packets are transmitted

according to these deadlines: those with earlier deadlines are transmitted first. In addition, it is also important to preserve FIFO order among transmitted packets in some data streams, e.g., in TCP connections. Keeping the relative order among packets in such connections is necessary in order to avoid a false activation of Fast Retransmit / Fast Recovery algorithms, which may cause lower transmissions rates and reduced throughputs [4].

To summarize, the BS needs to schedule its packets for transmission on the Downlink matrix according to the following requirements: (i) packets must be transmitted in a given rigid order, e.g., FIFO, and (ii) every packet transmission requires a rectangular set of matrix cells.

Allocation of rectangular cell sets for each packet is a resource management problem for which the 802.16 standard specifies no algorithm. Any vendor that implements an 802.16 system is expected to implement its own allocating algorithm. In this paper we investigate the complexity of this problem and develop an algorithm adhering to the aforementioned two requirements.

Problem definition. We investigate the *sequential rectangle placement problem* (SRP, for short), in which the input consists of:

Matrix: The (scheduling) matrix is given by its dimensions: $L \times H$. Without loss of generality we assume that $L \geq H$. We also define $S \triangleq L \cdot H$.

Jobs: The input sequence of jobs $N = J_1, J_2, \ldots, J_n$. Each job J_i is associated with a *size* $r_i \in \mathbb{N}$ and a non-negative weight w_i.

A solution is a *placement* which is an assignment of a rectangular set of matrix cells for each job in some *prefix* of N, where the number of matrix cells assigned for J_i is not smaller than r_i, and all rectangles are *non-intersecting*. We assume, without loss of generality, that the sum of all the job sizes is at most $L \times H$. The goal of the sequential rectangle placement problem is to find a placement of the longest possible prefix of jobs of the input sequence. In other words, our goal is to find a placement for the jobs $J_1, \ldots, J_t$, where t is as high as possible.

Given a placement of the prefix $J_1, \ldots, J_t$, the sum $\sum_{i=1}^{t} w_j$ is referred to as the weight of the placement. We evaluate the placement by its weight. In the sequel we prove that SRP is NP-hard, and in general is also hard to approximate. Therefore, we turn to look for approximation algorithms for some special cases. We consider two weight functions: the *unit* weight function, $w_i = 1$ for every job J_i, and the *proportional* weight function, $w_i = r_i$ for every job J_i. In the unit weight function the weight of the placement is simply the number of jobs that were placed on the matrix, while in the second the weight of the placement is the total area that is occupied by the placed jobs. The motivation for these two weight functions is as follows. In the unit weight function we count the number of served jobs, which is important in order to serve as many clients in the system as possible. In the proportional weight function we evaluate the amount of transmission resources we use (cells in the transmission matrix). This weight function is important because the wireless transmission resource is a very scarce resource that must be used as efficiently as possible [5].

A *bounding rectangle* of job J_i is a rectangular set of cells whose area is at least r_i, and that does not contain any proper sub rectangle whose area is at least r_i. That is, a bounding rectangle is a rectangle of length L_i and height H_i such that (1) $r_i \leq L_i H_i$, (2) $r_i > L_i(H_i - 1)$, and (3) $r_i > (L_i - 1)H_i$. Henceforth, we assume without loss of generality that a rectangle that is reserved for a job is a bounding rectangle. Notice that a bounding rectangle can be larger than the job size. For instance, a bounding rectangle for a job of size 5 can be of size 6, with length 3 and height 2. In this example, the bounding rectangle occupies one additional cell above the minimal number of matrix cells needed for placing this job. Thus, it wastes resources because free cells in a bounding rectangle cannot be used by other jobs [1]. On the other hand, the somewhat weakened requirement to use bounding rectangles yields some additional freedom in rectangle shaping, e.g., in the above example we are not restricted to a rectangle of dimensions 5×1 and can also use a rectangle of dimensions 3×2.

Related work. As far as we know this is the first paper to consider SRP. However, the following *rectangle packing problem* was considered by several studies: Given a set of rectangles $R_i = (a_i, b_i)$, for $i = 1, \ldots, n$, where a_i and b_i are the length and height of R_i, and a larger rectangle $R = (a, b)$. Each rectangles has a profit p_i. The goal is to pack a subset of the rectangles into R such that the total profit of packed rectangles is maximized. The packed rectangles may not overlap. This problem in NP-hard since it contains *knapsack* as the special case in which $a_i = a$ for every i. Constant factor approximation algorithms for this problem were given in [6,7], and a PTAS that packs the rectangles into a rectangle that is slightly bigger than R was presented in [8].

Note that SRP is very different from this rectangle packing problem. First, in SRP the requirement that the output placement is computed for a *prefix* of the input job sequence is crucial, while in the latter problem the input is an unordered set and the only optimized parameter is the total weight of the successfully placed rectangles. Furthermore, although in some studies of the rectangle packing problem, the given rectangles may be rotated (see, e.g., [6]) the dimensions of the rectangles are predetermined, while in SRP the input consists of requests for areas only.

Our results. We present both negative and positive results. On the negative side we prove that SRP is very hard to approximate. Specifically, we show that it is NP-hard to approximate SRP within a factor of $O(n^{1-\varepsilon})$ even when restricted to the case of unit weights. We also show that it is NP-hard to approximate SRP within a factor of $\frac{\sqrt{S}-1}{2}$ or within a factor of cn^2 for some constant c, even when restricted to the case of proportional weights. The full version of the paper contains two other hardness results. It is shown that SRP is NP-hard even when $r_i \leq \frac{1+\varepsilon}{n} \cdot S$ for every i, for any constant $\varepsilon > 0$. Moreover, we show that for this special case of SRP (with arbitrary weights) there cannot exist an r-approximation algorithm for any r, unless P=NP. All our hardness results are obtained using reductions from the well known NP-hard *partition problem* (PARTITION) [9].

On the positive side we present an $O(n \log n \log H)$ time approximation algorithm for SRP with proportional weights for the special case where $r_i \leq \beta S$ for some $\beta \in (0,1)$. The approximation ratio of the algorithm is $1 - \frac{\log H}{H} - \beta - \varepsilon$, for every $\varepsilon > 0$. Since $\frac{\log H}{H} \leq \frac{1}{2}$, the approximation ratio is not worse than $\frac{1}{2} - \beta - \varepsilon$, for every $\varepsilon > 0$. Our algorithm works as follows. First, it divides the range of job sizes into $O(\log H)$ subranges. For each such size subrange, the jobs in this subrange are further divided into jobs sets such that each set holds $O(\sqrt{L})$ jobs. The jobs in each such set are placed on a separate set of matrix rows. Furthermore, they are placed such that the unused area for each set is relatively small. For each prefix of the input job sequence, the algorithm uses Binary Search to compute the minimal number of matrix rows on which the prefix jobs can be placed. The algorithm uses Binary Search once again to determine the maximal prefix whose jobs can be placed on H matrix rows. In the analysis of our algorithm we compare the solution obtained by the algorithm to the area of the matrix, S, which is an upper bound on the weight of an optimal solution.

2 Hardness Results

We present four hardness results. We show that it is NP-hard to approximate SRP within a factor of $O(n^{1-\varepsilon})$ even for the case of unit weights. We also show that it is NP-hard to approximate SRP within a factor of $\frac{\sqrt{S}-1}{2}$ or within a factor of cn^2 for some constant c, even in the case of proportional weights. The full version of the paper contains two other hardness results. It is shown that SRP is NP-hard even when $r_i \leq \frac{1+\varepsilon}{n} \cdot S$ for every i, for any constant $\varepsilon > 0$. Moreover, we show that for this special case of SRP there cannot exist an r-approximation algorithm for any r, unless P=NP.

We present a reduction from PARTITION to SRP. Intuitively, given a PARTITION instance the reduction works as follows. First, it creates m jobs whose sizes are inflated versions of the PARTITION numbers. It produces a square matrix whose length (or height) is a bit more than half of the sum of the inflated numbers. Then, it adds a huge job whose dimensions are $(L-1) \times (H-1)$. The role of this job is to make sure that the other jobs use only one row and one column. The last set of jobs contains jobs of size one.

Reduction 1. *Let $(x_1, \ldots, x_m)$ be a* PARTITION *instance, and denote $B \triangleq \frac{1}{2}\sum_{j=1}^{m} x_i$. (Henceforth we assume that B is integral.) We construct an instance $(r_1, \ldots, r_n, L, H)$ as follows. First, let m' be an even integer. (The exact value of m' will be determined later.) We define $b = m'(B + \frac{1}{2})$ and $L = H = b + 1$. Also, let $n = 2 + m + m'$, and let $r_j = m' \cdot x_j$ for every $1 \leq j \leq m$, $r_{m+1} = b^2$, and $r_j = 1$ for every $m + 2 \leq j \leq n$. We refer to jobs $J_1, \ldots, J_m$ as* medium *jobs and to jobs $J_{m+2}, \ldots, J_n$ as* small *jobs.*

Note that the reduction is polynomial in case $m' = O(m^k)$ for some constant k.

Observation 1. $\sum_{j=1}^{n} r_j = L \cdot H$.

We show that if $(x_1, \ldots, x_m)$ belongs to PARTITION then all jobs can be placed. Otherwise, we will not be able to place more than m jobs.

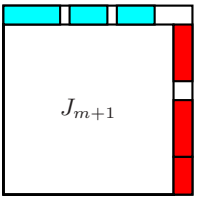


Fig. 3. Example of placement of J_{m+1}

Lemma 1. $(x_1, \ldots, x_m) \in$ PARTITION *if and only if the jobs* $J_1, \ldots, J_n$ *can be placed on an* $L \times H$ *matrix. Furthermore, if* $(x_1, \ldots, x_m) \notin$ PARTITION *then jobs* $J_1, \ldots, J_{m+1}$ *cannot be placed on the matrix.*

Proof. First, observe that since $r_{m+1} = b^2$, job J_{m+1} must be placed in such a way that leaves one empty column and one empty row, if it is placed in an $L \times H$ matrix. Hence, every other job must be placed as a rectangle of either length 1 or height 1. It follows that, if job J_{m+1} is placed on the matrix then the only freedom we have in deciding how to place a job $J_j \neq J_{m+1}$ is whether to place it in the empty row or in the empty column. (See example in Fig. 3).

If $(x_1, \ldots, x_m) \in$ PARTITION then there exists a subset $I \in \{1, \ldots, m\}$ such that $\sum_{i \in I} x_i = B$. Hence, we can place the medium jobs on the matrix as follows. If $j \in I$ we place the job in a rectangle of length r_j and height 1 in the empty row of the matrix, and otherwise we place the job in a rectangle of length 1 and height r_j in the empty column of the matrix. We can place all the medium jobs in I (not in I) in a row (column) since $B < b$. The small jobs can be placed in the remaining space due to Obs. 1. It follows that if $(x_1, \ldots, x_m) \in$ PARTITION, then all jobs can be placed on the matrix.

Now, for the other direction, let $(x_1, \ldots, x_m) \notin$ PARTITION and assume that jobs $J_1, \ldots, J_{m+1}$ are placed in the matrix. Denote by S_1 the set of medium jobs whose rectangle is placed in the empty row (the row that is left empty after the placement of job J_{m+1}), and denote by S_2 the set of medium jobs whose rectangle is placed in the empty column. We now claim that it follows that $\sum_{j \in S_1} x_j = \sum_{j \in S_2} x_j = B$. Notice that if this is not the case then $|\sum_{j \in S_1} x_j - \sum_{j \in S_2} x_j| \geq 2$, and therefore $|\sum_{j \in S_1} r_j - \sum_{j \in S_2} r_j| \geq 2m'$. Without loss of generality let $\sum_{j \in S_1} r_j > \sum_{j \in S_2} r_j$. Hence, since $\sum_{j=1}^{m} r_j = m' \cdot 2B$ we get that $\sum_{j \in S_1} r_j \geq m' \cdot B + m' = m'(B+1)$ and $\sum_{j \in S_2} r_j \leq m' \cdot B - m' = m'(B-1)$. It follows that $\sum_{j \in S_1} r_j > m'(B + \frac{1}{2}) + 1 = b + 1 = L$, a contradiction. □

The lemma directly implies that SRP is NP-hard. Next, we show that this problem is also very hard to approximate. In the sequel we denote the optimum of an SRP instance by OPT.

Theorem 1. *It is NP-hard to approximate* SRP *within a factor of* $\frac{\sqrt{S}-1}{2}$ *or within a factor of* cn^2 *for some constant* c*, even for the restricted case of proportional weights.*

Proof. We use Reduction 1 by letting m' be the first even number that is greater than or equal to m. By Lemma 1 if $(x_1, \ldots, x_m) \in$ PARTITION then all jobs can be placed, and the weight of the placement is $(b+1)^2$. On the other hand, if $(x_1, \ldots, x_m) \notin$ PARTITION, then the weight of any placement is at most $(b+1)^2 - b^2 = 2b+1$. Hence, for instances that are generated by Reduction 1 with proportional weights, distinguishing between instances for which OPT $= (b+1)^2$ and instances for which OPT $\leq 2b$ is NP-hard.

Suppose that there is a polynomial algorithm that computes $\frac{b}{2}$-approximate solutions for SRP. Then, we can use it on instances of Reduction 1 to determine whether OPT $= (b+1)^2$ and thus solve PARTITION. If OPT $= (b+1)^2$ then the algorithm computes a placement whose weight is at least $2b+4$. Otherwise, the algorithm computes a placement of weight at most $2b$. We therefore conclude that there cannot exist a $\frac{b}{2}$-approximation algorithm for SRP, unless P=NP.

It remains to relate $\frac{b}{2}$ to area of the matrix S and to the number of jobs n. Since $b = \sqrt{S} - 1$ it follows that it is NP-hard to approximate SRP within a factor of $\frac{\sqrt{S}-1}{2}$. By definition $b = m'(B + \frac{1}{2})$, hence $\frac{b}{2} \geq \frac{m}{2}(B + \frac{1}{2})$. Since $B \geq m$ and $n = \Theta(m)$, it follows that there exists a constant c such that $\frac{b}{2} \geq cn^2$. □

Theorem 2. *It is NP-hard to approximate* SRP *within a factor of* $O(n^{1-\varepsilon})$ *for every* $\varepsilon > 0$*, even for the restricted case of unit weights.*

Proof. For a given ε, we use Reduction 1 by setting $k = \left\lceil \frac{2}{\varepsilon} \right\rceil$ and letting m' be the first even number that is greater than or equal to m^k. The reduction is polynomial since k is a constant. In Lemma 1 we showed that if $(x_1, \ldots, x_m) \in$ PARTITION then all $n = m + m' + 2$ jobs can be placed on the matrix, and that otherwise at most m jobs can be placed on the matrix. Thus, for instances that are generated by Reduction 1 with unit weights, distinguishing between instances for which OPT $= n$ and instances for which OPT $\leq m$ is NP-hard.

Suppose that there is a polynomial algorithm that computes $O(m^{k-2})$-approximate solutions for SRP. Then, we can use it on the instances of Reduction 1 to determine whether OPT $= \Theta(m^k)$ and thus solve PARTITION. If OPT $= \Theta(m^k)$ then the algorithm computes a placement that places $\Omega(m^2)$ jobs. Otherwise, the algorithm outputs a placement containing at most m jobs. We conclude that there cannot exist an $O(m^{k-2})$-approximation algorithm for SRP, unless P=NP. It remains to relate m^{k-2} to the instance size n:

$$m^{k-2} = \Theta(n^{\frac{k-2}{k}}) = \Theta(n^{1-\frac{2}{k}}) = \Omega(n^{1-\frac{2}{\lceil 2/\varepsilon \rceil}}) = \Omega(n^{1-\varepsilon})$$

and we are done. □

3 An Algorithm for SRP with Proportional Weights

In this section we develop an approximation algorithm for SRP with proportional weights. For the special case in which $r_i \leq \beta \cdot S$ for every job J_i where $\beta \in (0,1)$, the algorithm achieves an approximation ratio of $1 - \frac{\log H}{H} - \beta - \varepsilon$, for any $\varepsilon > 0$.

Definitions and Notation. Before presenting the algorithm, we introduce some notation. The placement computed by our algorithm is *row oriented*.

Definition 1. *A job placement of a job set JS is called* row oriented *if the following conditions hold:*

1. *The set JS is divided into some disjoint subsets $\{JS_i\}_{i=1}^{k}$.*
2. *The jobs in each job set JS_i are placed on a set of consecutive matrix rows* dedicated *to JS_i. The rows occupied by distinct job sets are distinct.*
3. *For any job set JS_i, and for any job $J_k \in JS_i$, the base of the bounding rectangle of J_k is placed on the first row dedicated to JS_i and the height of the bounding rectangle is equal to the number of rows dedicated to JS_i.*

Let P be a row oriented placement and let JS_i be a job set of P. The number of rows *required by* JS_i, denoted by $\text{ROWS}(JS_i)$, is the minimal number of rows on which all jobs of JS_i can be placed adhering to Def. 1. The number of rows *required by* P, denoted by $\text{ROWS}(P)$, is the total number of rows required by the job sets of P.

Our algorithm maintains a collection of disjoint job sets, where the sizes of all jobs in each set are within a certain range. The ranges for the job sets depend on L and H, the matrix dimensions, and are defined as follows.

Definition 2. *We divide the jobs into three types*[1]*:*

- **Small jobs***: A job J_k is called* small *if $r_k \leq 2\sqrt{L}$.*
- **Medium jobs***: A job J_k is called* medium *if $2\sqrt{L} < r_k \leq \frac{H}{2}\sqrt{L}$. For each i, $2 \leq i \leq \log H - 1$, a medium job is called i-*medium *if $2^{i-1}\sqrt{L} < r_k \leq 2^i\sqrt{L}$.*[2]
- **Large jobs***: A job J_k is called* large *if $r_k > \frac{H}{2} \cdot \sqrt{L}$.*

We consider row oriented placements in which for each type of a job set, the number of rows required by the set is limited.

Definition 3. *A row oriented placement P is called* bounded *if the jobs sets in P follow Def. 2 and the number of rows required by each job set is bounded as follows:*

- **Small jobs:** *A job set of small jobs can have at most one row.*
- **Medium jobs:** *A job set of i-medium jobs ($2 \leq i \leq \log H - 1$) can have at most 2^i rows.*
- **Large jobs:** *A job set of large jobs can have at most $2H$ rows. (Note that the value $2H$ is used for intermediate computations. The output placement P satisfies: $\text{ROWS}(P) \leq H$.)*

The maximal number of matrix rows allowed for JS_i is denoted by $\text{MAX}(JS_i)$.

The Algorithm. We present an algorithm called **Placement** that uses a procedure called **Prefix**. We first describe the procedure and then describe the algorithm. Procedure **Prefix** gets as input the matrix dimensions, L and H,

[1] The distinction between job sizes is different from the one that was made in Sect. 2.
[2] We assume that $\log H$ is integral for reasons of clarity.

and a finite sequence of jobs, N_f, and computes a row oriented placement P_f. Placement P_f computed by Procedure **Prefix** includes *all jobs* in N_f, while $\textsc{rows}(P_f)$ may exceed H. A job set of P_f can be either *open* or *closed*, where jobs can be added only to open job sets. At any given time during execution of Procedure **Prefix** there exists a single open job set for every range of job sizes, specified by Def. 2. Therefore, at any given time, the number of non-empty open sets is at most $\log H$.

Procedure **Prefix** works as follows:

1. Divide N_f into $\log H$ disjoint sets according to Def. 2.
2. Open a job set M_1^S of one matrix row and place small jobs from N_f, one after the other, in M_1^S. If the single row of M_1^S is depleted, close M_1^S, add it to P_f and start a new job set, M_2^S, with a single row. Continue in this fashion until all small jobs in N_f are placed. Add the last set, whose row may not be full, to P_f.
3. For each i, where $2 \leq i \leq \log H - 1$, open a job set M_1^i of 2^i matrix rows and place i-medium jobs from N_f, one after the other, to M_1^i. This is done in the following fashion: sequentially start accumulating the sum $\sum_j \left\lceil \frac{r_j}{2^i} \right\rceil$, where j runs over the sequence of i-medium jobs, until (and not including) the job that causes the sum to exceed L. At this point, the jobs in M_1^i are determined, it is closed and $\textsc{rows}(M_1^i)$ is set to be $\textsc{max}(M_1^i) = 2^i$. Open a new job set M_2^i for i-medium jobs with 2^i rows. Continue in this fashion until all i-medium jobs in N_f are divided into job sets, all of which, except perhaps the last one, are closed.
4. Open a single job set, M^L, of $2H$ rows, for large jobs, place all large jobs of N_f in M^L, and add it to P_f. Notice that since the sum of the sizes of all the jobs in the input is bounded by $L \cdot H$, a set of area $2H \cdot L$ is sufficient to accommodate all the large jobs in the input sequence.
5. For each open job set of i-medium jobs, M_k^i, use binary search to compute $\textsc{rows}(M_k^i)$. In each iteration we simply set a tentative value for $\textsc{rows}(M_k^i)$ and check whether this value is sufficient to place all jobs in M_k^i. When this process ends, the right value for $\textsc{rows}(M_k^i)$ is known, and M_k^i is added to P_f. Use the same process to compute $\textsc{rows}(M^L)$.
6. Compute the total number of rows required for P_f.

Given the above description of Procedure **Prefix**, Algorithm **Placement** works as follows. First, it finds, using binary search on the input sequence of n jobs, what is the maximum number of jobs that can be placed on the matrix. Every placement trial is done by Procedure **Prefix**. Let $J_1, \ldots, J_t$ be longest prefix of the input job sequence for which $\textsc{rows}(P) \leq H$, where P is the placement computed by Procedure **Prefix**. The algorithm returns the placement P.

Analysis. We start the analysis by bounding the running time of the algorithm.

Theorem 3. *The running time of Algorithm* ***Placement*** *is* $O(n \cdot \log n \cdot \log H)$.

Proof. We first deal with Procedure **Prefix**. Assume that it is executed with input sequence of n' jobs. The running time of all stages of the procedure apart

from the fifth stage is $O(n')$. It remains to bound the complexity of computing the number of rows required for the sets of medium and large jobs. Let J_j be an i-medium job that is placed in job set M_k^i (a large job that is placed in M^L, respectively). During the fifth stage, J_j is considered at most $\log 2^i$ times due to the binary search for $\text{ROWS}(M_k^i)$. It follows that the time complexity of computing $\text{ROWS}(M_k^i)$ is $O(|M_k^i| \cdot \log 2^i) = O(|M_k^i| \log H)$. A similar argument shows that the time complexity of computing $\text{ROWS}(M^L)$ is $O(|M^L| \log H)$. Hence the total running time of the fifth stage is $O(n' \log H)$.

Since the complexity of the procedure is $O(n' \log H)$, and since the procedure is invoked $O(\log n)$ times with an input sequence of size $n' \leq n$, the complexity of the algorithm is $O(n \cdot \log n \cdot \log H)$. □

Assume that Procedure **Prefix** succeeded in placing jobs $J_1, \ldots, J_t$, but failed to place jobs $J_1, \ldots, J_{t+1}$. We compute the wasted matrix space of the placement P of jobs $J_1, \ldots, J_t$. Observe that the space wasted by the P comprised of two parts. First, there is the space that is wasted within the rows that P uses. The second part is the space that P does not use because this part is not large enough to accommodate J_{t+1}. We first bound the wasted space due to open job sets and due to closed job sets. Then, we calculate the overall wasted space within the rows that P uses. Afterwards, we bound the space that is not used by P. We show that the total wasted space is at most $(\frac{\log H}{H} + \frac{3}{\sqrt{L}} + \beta) \cdot S$.

Lemma 2. *Let M be a job set of medium or large jobs. Then, $|M| \leq 2\sqrt{L}$.*

Proof. First, assume that M is a set of i-medium jobs. By Def. 3, the number of rows in M is at most 2^i. Since the size of the smallest job in M is at least $2^{i-1}\sqrt{L}$, the length of the base of each job in M is at least $\frac{2^{i-1} \cdot \sqrt{L}}{2^i} \geq \frac{\sqrt{L}}{2}$. Since the base of the bounding rectangle of each job resides on the first matrix row dedicated to M, the number of jobs in M is not greater than $\lfloor \frac{L}{\sqrt{L}/2} \rfloor \leq 2\sqrt{L}$.

Assume that M is the (single) set of large jobs. Since the sum of the sizes of the jobs in the input is at most $L \cdot H$, and the size of a large job is at least $\frac{H}{2}\sqrt{L}$, there can be at most $2\sqrt{L}$ large jobs. □

Let $A_w(M)$ be the wasted area caused by a job set M. We first bound the waste in open sets.

Lemma 3. *Let M be an open set in P. Then, $A_w(M) < L + 2\sqrt{L}(\text{ROWS}(M) - 1)$.*

Proof. If $\text{ROWS}(M) = 1$ then we are done. Hence, the lemma is immediate for small jobs. Assume that M is either i-medium or large and that $\text{ROWS}(M) > 1$. Since the jobs in M do not fit in $\text{ROWS}(M) - 1$ rows it follows that

$$\sum_{J_j \in M} [r_j + (\text{ROWS}(M) - 2)] > L \cdot (\text{ROWS}(M) - 1)$$

where $\text{ROWS}(M) - 2$ is the maximum wasted space per job, if we use only $\text{ROWS}(M) - 1$ rows. Hence,

$$A_w(M) = L \cdot \text{ROWS}(M) - \sum_{J_j \in M} r_j < L + |M|(\text{ROWS}(M) - 2) \ .$$

$|M| \leq 2\sqrt{L}$ by Lemma 2, hence $A_w(M) < L + 2\sqrt{L}(\text{ROWS}(M) - 1)$. □

Next, we bound the waste in closed sets.

Lemma 4. *Let M be a closed set. Then, $A_w(M) \leq \text{ROWS}(M) \cdot 3\sqrt{L}$.*

Proof. The lemma is immediate for small jobs. Also, notice that there is no closed set of large jobs. Assume that M consists of i-medium jobs and recall that by the algorithm, $\text{ROWS}(M) = \text{MAX}(M)$. Let J_q be the job that was rejected from M just before M was closed. Hence,

$$\sum_{J_j \in M}[r_j + (\text{MAX}(M) - 1)] + r_q > \text{MAX}(M) \cdot L$$

where $\text{MAX}(M) - 1$ is the maximum wasted space per job. Thus,

$$A_w(M) = \text{MAX}(M) \cdot L - \sum_{J_j \in M} r_j < (\text{MAX}(M) - 1) \cdot |M| + r_q$$

Since $|M| \leq 2\sqrt{L}$ by Lemma 2, and since $r_q \leq 2^i\sqrt{L} = \text{MAX}(M)\sqrt{L}$, we get that $A_w(M) < \text{ROWS}(M) \cdot 2\sqrt{L} + \text{ROWS}(M) \cdot \sqrt{L} < \text{ROWS}(M) \cdot 3\sqrt{L}$ as required. □

Lemma 5. *The waste within rows used by P is at most $L \log H + 3\sqrt{L}\text{ROWS}(P)$.*

Proof. By Lemma 3, the wasted space incurred by the open set M^i of i-medium jobs, or by the open set of large jobs, M^L, is at most $L + 2\sqrt{L} \cdot (\text{ROWS}(M^i) - 1)$. Hence, the total wasted space incurred by open job sets is at most:

$$\sum_{i=1}^{\log H}[L + 2 \cdot \sqrt{L} \cdot (\text{ROWS}(M^i) - 1)] < L \cdot \log H + \text{OPEN}(P) \cdot 2\sqrt{L}$$

where $\text{OPEN}(P)$ denotes that number of rows dedicated to open sets.

By Lemma 4, the wasted space incurred by each closed job set M of P is at most $\text{ROWS}(M) \cdot 3\sqrt{L}$. Thus, the total wasted space incurred by all closed job sets is at most:

$$\sum_M \text{ROWS}(M) \cdot 3\sqrt{L} \leq \text{CLOSED}(P) \cdot 3\sqrt{L}$$

where the summation is taken over all closed sets and $\text{CLOSED}(P)$ denotes that number of rows dedicated to close sets.

Since $\text{ROWS}(P) = \text{OPEN}(P) + \text{CLOSED}(P)$, it follows that

$$A_w(P) < L \cdot \log H + \text{OPEN}(P) \cdot 3\sqrt{L} + \text{CLOSED}(P) \cdot 3\sqrt{L} < L \cdot \log H + 3\sqrt{L} \cdot \text{ROWS}(P)$$

The lemma follows. □

Theorem 4. *Let $L \geq 9$. Then, the wasted space of the solution computed by Algorithm **Placement** is at most $L \log H + 3H\sqrt{L} + \beta \cdot S$.*

Proof. Consider the invocation of Procedure **Prefix** on $J_1, \ldots, J_{t+1}$ and let P' be the computed placement of this invocation. Clearly, $\text{ROWS}(P') > H$. By lemma 5, the space wasted by P' satisfies:

$$A_w(P') = L \cdot \text{ROWS}(P') - \sum_{j=1}^{t+1} r_j < L \cdot \log H + 3\sqrt{L} \cdot \text{ROWS}(P')$$

Hence, the total matrix space wasted by P satisfies:

$$\begin{aligned} A_w(P) &= L \cdot H - \textstyle\sum_{j=1}^{t} r_j \\ &= L \cdot \textsc{rows}(P') - L \cdot (\textsc{rows}(P') - H) - \textstyle\sum_{j=1}^{t+1} r_j + r_{t+1} \\ &< L \log H + 3\sqrt{L} \cdot \textsc{rows}(P') - L \cdot (\textsc{rows}(P') - H) + r_{t+1} \\ &= L \log H + 3H\sqrt{L} + 3\sqrt{L}(\textsc{rows}(P') - H) - L(\textsc{rows}(P') - H) + r_{t+1} \\ &= L \log H + 3H\sqrt{L} + (\textsc{rows}(P') - H) \cdot (3\sqrt{L} - L) + r_{t+1} \end{aligned}$$

If $L \geq 9$ it follows that $A_w(P) < L \log H + 3H\sqrt{L} + \beta \cdot S$. □

Corollary 1. *Let $L \geq 9$. Then, the approximation ratio of Algorithm **Placement** is $1 - \frac{\log H}{H} - \frac{3}{\sqrt{L}} - \beta$.*

Proof. Let OPT be the area of the matrix that is covered by an optimal placement, and let ALG be the area covered by P. Then by Theorem 4 it follows that $\frac{\text{ALG}}{\text{OPT}} \geq \frac{S - A_w(P)}{S} > \frac{S - L\log H - 3H\sqrt{L} - \beta \cdot S}{S} \geq 1 - \frac{\log H}{H} - \frac{3}{\sqrt{L}} - \beta$. □

Since $L \geq H$ we can solve the problem exactly for small L's using exhaustive search. Hence, the approximation ratio is in fact $1 - \frac{\log H}{H} - \beta - \varepsilon$, for any $\varepsilon > 0$. Notice that $\frac{\log H}{H} \leq \frac{1}{2}$, thus the ratio is not worse than $\frac{1}{2} - \beta - \varepsilon$, for any $\varepsilon > 0$.

Acknowledgment. We thank Yaron Alpert for introducing the problem to us.

References

1. IEEE Standard for Local and Metropolitan Area Networks: IEEE 802.16e, Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Ssytems (2005)
2. Nordboten, A.: LMDS systems and their application. IEEE Communications Magazine 38(6), 150–154 (2000)
3. Ferrari, D., Verma, D.C.: A scheme for real-time channel establishment in wide-area networks. IEEE Journal on Selected Areas in Communications 8(3), 368–379 (1990)
4. Allman, M., Paxson, V., Stevens, W.: TCP congestion control. RFC 2581 (1999)
5. Nicopolitidis, P., Obaidat, M.S., Papadimitriou, G.I., Pomportsis, A.S.: Wireless Networks. John Wiley & Sons Ltd, Chichester (2003)
6. Jansen, K., Zhang, G.: On rectangle packing: Maximizing benefits. In: 15th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 204–213. ACM Press, New York (2004)
7. Jansen, K., Zhang, G.: Maximizing the number of packed rectangles. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 362–371. Springer, Heidelberg (2004)
8. Fishkin, A.V., Gerber, O., Jansen, K., Solis-Oba, R.: Packing weighted rectangles into a square. In: Jedrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 352–363. Springer, Heidelberg (2005)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York (1979)

Shorter Implicit Representation for Planar Graphs and Bounded Treewidth Graphs

Cyril Gavoille* and Arnaud Labourel

LaBRI, Université de Bordeaux, France
{gavoille,labourel}@labri.fr

Abstract. Implicit representation of graphs is a coding of the structure of graphs using distinct labels so that adjacency between any two vertices can be decided by inspecting their labels alone. All previous implicit representations of planar graphs were based on the classical three forests decomposition technique (a.k.a. Schnyder's trees), yielding asymptotically to a $3\log n$-bit[1] label representation where n is the number of vertices of the graph.

We propose a new implicit representation of planar graphs using asymptotically $2\log n$-bit labels. As a byproduct we have an explicit construction of a graph with $n^{2+o(1)}$ vertices containing all n-vertex planar graphs as induced subgraph, the best previous size of such induced-universal graph was $O(n^3)$.

More generally, for graphs excluding a fixed minor, we construct a $2\log n + O(\log\log n)$ implicit representation. For treewidth-k graphs we give a $\log n + O(k\log\log(n/k))$ implicit representation, improving the $O(k\log n)$ representation of Kannan, Naor and Rudich [18] (STOC '88).

Our representations for planar and treewidth-k graphs are easy to implement, all the labels can be constructed in $O(n\log n)$ time, and support constant time adjacency testing.

1 Introduction

How to represent a graph in memory is a basic and fundamental data structure question. The two basic ways are adjacency matrices and adjacency lists. The latter representation is space efficient for sparse graphs, but adjacency queries require searching in the list, whereas matrices allow fast queries to the price of a super-linear space. Another technique, called *implicit representation* or *adjacency labeling scheme*, consists in assigning distinct labels to each vertex such that adjacency queries can be computed alone from the labels of the two involved vertices without any extra information source. So the graph can be manipulated by keeping only its labels in memory, any other global information on the graph (like its matrix) can be removed. The goal is to minimize the maximum length of a label associated with a vertex while keeping fast adjacency queries.

* Both authors are supported by the ANR-projects GEOCOMP and GRAAL.

[1] All logarithms are in base two.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 582–593, 2007.

The notion of adjacency labeling scheme is closely related to that of induced-universal graphs, introduced by [12] for VLSI circuit design. A graph $\mathcal{U}$ is said to be *induced-universal* for a given graph family $\mathcal{F}$, if every graph of $\mathcal{F}$ is isomorphic to some induced subgraph of $\mathcal{U}$. Kannan, Naor and Rudich [18] established that there is an adjacency labeling scheme with labels of k bits for $\mathcal{F}$ if and only if there exists an induced-universal graph with 2^k vertices. Therefore, the combinatorial problem of constructing a small induced-universal graph for $\mathcal{F}$ is equivalent in designing a labeling scheme with short labels for every graph of $\mathcal{F}$, any result on one of the problems transferring on the other. For instance, the best known induced-universal graph for the class of n-node forests, namely $n \cdot 2^{O(\log^* n)}$, is actually derived from a labeling scheme with $\log n + O(\log^* n)$-bit[2] labels [3].

We note that a labeling scheme transfers to an effective construction of the corresponding induced-universal graph: the vertex-set of the universal graph is the set of all possible labels, and an edge is added if the adjacency test between the two corresponding vertices/labels is positive. However, converting a small induced-universal graph into a compact labeling scheme does not give, in general, any efficient representation in term of computation of all the labels for a graph and adjacency testing time.

1.1 Related Works

Motivated by applications in XML search engines, and distributed applications as peer-to-peer overlay networks or network routing, several queries on distributed data-structures have been investigated recently. In this framework, distributed data-structure can be seen as a label assigned to each node such that queries can be answered by inspecting the labels only, without any other source of information. For instance, address-based routing in trees [13,19,25], distance queries for interval and permutation graphs [5,14], sibling and ancestry in rooted trees [1], etc. have $O(\log n)$-bit distributed data-structures.

Representation of graphs with short labels, introduced by Breuer [8], have been investigated by Kannan, Naor and Rudich [18]. They construct adjacency labeling schemes for several families of graphs including treewidth[3]-k and arboricity[4]-α graphs using respectively $O(k \log n)$-bit and $(\alpha+1)\log n$-bit labels. In particular, this latter scheme gives an induced-universal graph for trees ($\alpha = 1$) of size $O(n^2)$, and combined with the linear time Schnyder's tree decomposition [24], this leads to the first effective $4 \log n$-bit labeling for planar graphs ($\alpha = 3$).

The size of the induced-universal graph for trees was later improved in a non-trivial way to $O(n \log n)$ by Chung [9]. This transfers to a non-constructive adjacency labeling with labels of $\log n + \log\log n + O(1)$ bits for trees, and more generally of $\alpha \log n + O(\alpha \log\log n)$ bits for arboricity-α graphs. This result has

[2] $\log^* n$ denotes the number of times log should be iterated to get a constant.

[3] The original result was stated for c-decomposable graphs which are exactly the graphs of treewidth $k = \Theta(c)$.

[4] Which is the smallest number of forests in which the edge-set of the graph can be partitioned.

been further improved by the use of an efficient labeling scheme to $\alpha \log n + O(\log^* n)$ bits [3], complemented with a $\alpha \log n - O(\alpha^2)$ lower bound. This best to date result yields to an adjacency labeling scheme for trees of $\log n + O(\log^* n)$. For planar graphs, the resulting $3 \log n + O(\log^* n)$-bit labeling can be slightly improved by observing that: 1) planar graphs can be decomposed into three forests those one is spanning and of bounded degree [17]; and 2) adjacency in bounded degree forests can be done efficiently in $\log n + O(1)$ [7]. Thus, a vertex can store its label in the bounded degree forest plus the label of its parent in each of the two other forests, leading to a $3 \log n + O(1)$ labeling with constant time adjacency. This converts into an induced-universal graph of size $O(n^3)$.

Finally, we remark that the best up to date planar representation essentially uses the three forests decomposition.

1.2 Our Contributions

1. Surpassing the three forests decomposition of planar graph, we propose a new $2 \log n + O(\log \log n)$ bit adjacency labeling scheme supporting constant query time.
2. More generally, for graphs excluding a fixed graph H as minor[5] (e.g., planar graphs exclude K_5), we propose a $2 \log n + O(h \log \log(n/h))$ bit labeling scheme where h is a constant only depending on H.
3. For treewidth-k graphs, a subclass of graphs excluding K_{k+2}-minor, we improve the label length to $\log n + O(k \log \log(n/k))$ supporting adjacency testing in constant time (independent of k).
4. We also show that every adjacency labeling scheme for treewidth-k graphs requires labels of $\log n + \Omega(k)$ bits, proving that the linearity dependency on k in the second order term of our scheme is necessary.
5. All these results transfer also to effective construction of smaller induced-universal graphs of size $n^{2+o(1)}$ for minor-free graphs, and of $n^{1+o(1)}$ for bounded treewidth graphs, previous bounds were $O(n^3)$ for planar graphs and $n^{O(k)}$ for treewidth-k graphs.

1.3 Techniques

In fact our results on planar and minor-free graphs are derived from our $(1 + o(1)) \log n$-bit label scheme for bounded treewidth graphs using suitable edge-partitions. Although planar graphs can be decomposed into three treewidth-1 graphs (forests), they can also be decomposed, in linear time as well, in two bounded treewidth graphs as follows: 1) partition the vertices into layers L_i at distance i from an arbitrary vertex; 2) the graph H_i, $i > 0$, induced by the edge $\{u, v\}$ with $u, v \in L_i$ or with $u \in L_i$ and $v \in L_{i-1}$ has bounded treewidth (actually at most three from [11]); 3) the two graphs composed as the union of the H_i's for respectively odd and even i are therefore of bounded treewidth since no edges lie between H_i and H_{i+2}.

[5] That are graphs for which H can be obtained by edge contractions and taking subgraphs.

From the above discussion, a λ-bit labeling for treewidth-k graphs yields a 2λ-bit labeling scheme for planar graphs, because it suffices to assign to each vertex the label for each subgraph and to test adjacency in the two subgraphs. Actually, it has been recently proved that planar graphs can be edge-partitioned into two outerplanars [16] (that are of treewidth two), and in linear time [17]. It has been proved a similar result for graphs excluding a fixed minor [10]: they can be edge-partitioned into two subgraphs of treewidth only depending on the excluded minor.

Therefore, our $\log n + O(\log\log n)$ labeling scheme for bounded treewidth graphs implies a $2\log n + O(\log\log n)$ not only for planar graphs but also for all minor-free graphs (including bounded genus graphs for instance). All previous schemes were based on the bounded arboricity of bounded treewidth or minor-free graphs. Unfortunately, arboricity-α graphs require labels of at least $\alpha\log n - O(\alpha^2)$ bits from the lower bound of [3], and outerplanar graphs (which exclude K_4) have already arboricity two. So there is no advantage of decomposing a graph into low arboricity subgraphs, unlike decomposing into low treewidth subgraphs: 2 treewidth-2 graphs is better than 3 treewidth-1 graphs.

From the above discussion, we detail our scheme only for treewidth-k graphs. A graph has *treewidth* k if and only if it is a subgraph of a chordal (or triangulated) graph[6] of maximum clique size $k+1$. Chordal graphs, and therefore treewidth-k graphs, have a natural clique tree structure, and they can also be defined as graphs having a Robertson-Seymour's tree-decomposition [23] in subgraphs of at most $k+1$ vertices, called *bags*. Informally, the set of bags forms a cover of the graph (each vertex and edge belongs to at least one bag), and they must be one-to-one mapped to the nodes of a tree T such that the set of bags containing a given vertex of the graph induces a connected component of T.

Let us first observe that an efficient labeling scheme on trees (like the one of [3]) does not transfer to a labeling scheme for graphs having some "good" tree-decomposition because bags may intersect in a non trivial way.

The next section present our labeling scheme. Due to lack of space the lower bound appears in the full version.

2 A Simple Adjacency Labeling Scheme

In this section we prove the main result of this paper that is:

Theorem 1. *The family of n-vertex treewidth-k graphs enjoys an adjacency labeling scheme with labels of $\log n + O(k\log\log(n/k))$ bits, and with a constant adjacency query time. Moreover, for every fixed k, all the labels of such a graph can be computed in $O(n\log n)$ time.*

The construction of the labels relies on finding a chordal supergraph with minimum maximal clique (i.e., $k+1$), whose the associated decision problem is NP-complete [4]. For each fixed k, linear time algorithms are known [6]. We

[6] I.e., a graph whose the length of the longest induced cycle is at most three.

also note that for trees, outerplanar and series-parallel graphs (i.e., for $k \leqslant 2$) efficient and simple algorithms exist, and our compact representation for planar graph is based on outerplanar graphs only.

As said previously, the results for the $2\log n + O(\log\log n)$ bit labeling for planar graphs, and more generally minor-free graphs, and the small induced-universal graphs, are simple corollaries of Theorem 1.

2.1 Preliminaries

Our scheme relies on the chordal representation: every treewidth-k graph is the spanning subgraph of a chordal graph, called *triangulation*, whose maximal cliques have no more than $k+1$ vertices. Chordal graphs of maximum clique size $k+1$ have two important properties. Firstly, there exists a clique whose its removal leaves the graph with connected components at most half the size of the original graph [15]. Secondly, there is an elimination ordering scheme of the vertices, called *simplicial elimination scheme*, such that the neighborhood of each just removed vertex is a clique in the remaining graph. Our scheme strongly relies on graphs having these both properties we formalize hereafter.

A graph G is *s-separable* if every subgraph H has a half-separator of size at most s, i.e., a subset of at most s vertices whose removal leaves H in connected components of at most $|V(H)|/2$ vertices. A graph is *k-clique orientable* if its edges can be oriented such that the out-neighborhood of every vertex induces a clique of size at most k. Such an orientation is called a k-clique orientation.

Definition 1. *A* (k,s)-triangulation *is a graph that is k-clique orientable and s-separable. A* (k,s)-graph *is a subgraph of a* (k,s)*-triangulation.*

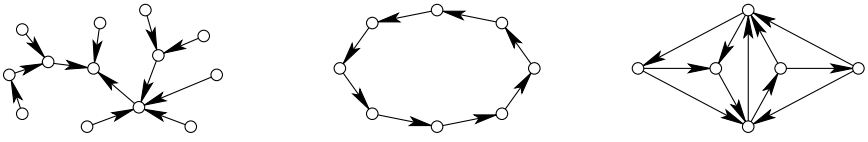


Fig. 1. A $(1,2)$-triangulation, a $(1,2)$-triangulation, and a $(1,2)$-triangulation

According to this definition, treewidth-k are $(k,k+1)$-graphs, since their triangulations are $(k+1)$-separable and the simplicial elimination scheme provides a k-clique orientation. However, forests are $(1,1)$-graphs whereas they are of treewidth one. Cycles (treewidth 2) are 1-clique orientable and 2-separable, and thus are $(1,2)$-graphs (they are also $(1,2)$-triangulations, see Fig. 1). We can also check that cliques of k vertices are $(\lceil k/2\rceil, \lceil k/2\rceil)$-graphs whereas they are of treewidth $k-1$. So, $(k,k+1)$-graphs are a strictly wider family of graphs than treewidth-k graphs.

A *k-complex* of a graph G is a subgraph K isomorphic to depth-1 tree with at most k leaves. Its *root*, denoted by $\mathrm{root}(K)$, is the non-leaf node of K. Two complexes K and K' of G are said *adjacent* if $\mathrm{root}(K) \neq \mathrm{root}(K')$, and if either

root$(K) \in V(K')$ or root$(K') \in V(K)$. Clearly, if two complexes are adjacent, then their roots are adjacent in G.

With each vertex u of a (k,s)-graph G one can associate a k-complex K_u of root u such that each edge of G is covered by exactly one k-complex, as follows: K_u is formed by u and its neighbors in G that are out-neighbors in the (k,s)-triangulation of G. Then, there is a trivial adjacency labeling scheme consisting in labeling u by a coding $V(K_u)$. It is clear that u and v are adjacent if and only if their associated complexes K_u and K_v are. This trivially yields to labels of $(k+1)\lceil \log n \rceil$ bits. We shall see that a much better implementation (or coding) of complexes can be achieved using the fact that, in the triangulation of G, K_u is a clique.

For this purpose we need a particular partition of the vertices of G.

Definition 2. *A* bidecomposition *of a graph G is a rooted binary tree T where nodes, called* parts*, form a partition of $V(G)$ such that the parts of the endpoints of every edge of G are related*[7] *in T.*

The part of T containing vertex u is denoted by T_u. A straightforward observation from the definition of T is that the parts of all the vertices of a subgraph K are pairwise related if K is a clique.

2.2 Finding a Suitable Bidecomposition

Lemma 1. *Let $\tilde{G}$ be an s-separable graph of n vertices. Then, $\tilde{G}$ has a bidecomposition such that the parts of depth h have at most $s\lfloor \log(2n/s) - h \rfloor$ vertices.*

Moreover, such a bidecomposition can be computed in $O(sn \log(n/s))$ time if we assume that a half-separator of size s for any subgraph of $\tilde{G}$ can be computed in time linear in the size of the subgraph.

Proof. Let $\tilde{G}$ be an s-separable graph with n vertices. Since every subgraph of an s-separable graph is also s-separable, we essentially need to construct the root of the willing bidecomposition T, i.e., to prove the result for depth $h = 0$, and repeat recursively the process on the remaining subgraphs. In the following, the root of the bidecomposition is called *biseparator*.

The root B of T (the biseparator) and the partition (V_1, V_2) of $V(\tilde{G}) \setminus B$ are computed thanks to the following simple procedure called Biseparator (Algorithm 1), where Half-Separator(H, s) is a subroutine computing a half-separator of size at most s for the graph H.

At each iteration of the while-main of Biseparator the size of H is divided by at least two since all the resulting components of $H \setminus S$ are removed from H, and the remaining largest component is of size at most $|V(H)|/2$. So, there is at most $\log(n/s)$ iterations.

At each step the size of B increases by at most s vertices, and the final step (last statement) adds at most s vertices to B. Therefore, $|B| \leqslant s \cdot \lfloor \log(n/s) \rfloor + s = s \cdot \lfloor \log(2n/s) - h \rfloor$ with $h = 0$.

[7] Two nodes of a rooted tree are *related* if one is ancestor of the other.

Algorithm 1. BISEPARATOR

Input : an s-separable graph $\tilde{G}$

Output: A biseparator B for $\tilde{G}$ and the associated vertex partition (V_1, V_2)

$H := \tilde{G}$; $V_1 := V_2 := B := \varnothing$
while $|V(H)| > s$ **do**
  $S :=$ HALF-SEPARATOR(H, s); $H := H \setminus S$; $B := B \cup S$
  forall *connected component* C *of* H *except the greatest* **do**
    $H := H \setminus C$;
    if $|V_1| > |V_2|$ **then** $V_2 := V_2 \cup V(C)$ **else** $V_1 := V_1 \cup V(C)$
$B := B \cup H$

In order to insure the correctness of the bidecomposition, we show the following loop invariant.

$$(P): \quad |V_1|, |V_2| \leqslant n/2 \quad \text{and} \quad V(H) \cup V_1 \cup V_2 \cup B = V(\tilde{G})$$

It is straightforward to verify that (P) is true at the beginning of the main loop. Let us show that (P) remains true at the end of the imbricated loop. The loop invariant clearly remains true after computation of the half-separator and the statement $H := H \setminus S$ and $B := B \cup S$. Assume w.l.o.g. that $|V_1| \geqslant |V_2|$. We have $V(H) \cup V_1 \cup V_2 \subseteq V(\tilde{G})$ since (P) is true. We obtain the following relation for the size of the sets.

$$\begin{aligned}
|V(H)| + |V_1| + |V_2| &\leqslant n \\
|V(H)| &\leqslant n - |V_1| - |V_2| \leqslant n - 2|V_2| \quad (|V_1| \geqslant |V_2|) \\
|V(H)|/2 &\leqslant n/2 - |V_2| \\
|V(C)| &\leqslant n/2 - |V_2| \text{ (there is yet another larger component in } H) \\
|V(C)| + |V_2| &\leqslant n/2
\end{aligned}$$

The property (P) remains true at the end of the imbricated loop. The property (P) is loop invariant and so is true at the end of the main loop. We have that $|V_1|, |V_2| \leqslant n/2$. Moreover, there is no edge linking vertices of V_1 to vertices of V_2 since what we add to V_1 or V_2 is a full connected component of H.

The bidecomposition T of $\tilde{G}$ is obtained by applying recursively the process on the graphs induced by V_1 and V_2. We then link to B (the root of T) the resulting bidecompositions if there are non-empty.

Since the size of V_1 and V_2 are $\leqslant n/2$, by induction, the size of their biseparators (so parts of depth $h = 1$ in T) would be at most $s \cdot \lfloor \log(2(n/2)/s)) \rfloor = s \cdot \lfloor \log(2n/s) - 1 \rfloor$. More generally, for an arbitrary depth $h \geqslant 0$, the parts are of size $s \cdot \left\lfloor \log(2(n/2^h)/s)) \right\rfloor = s \cdot \lfloor \log(2n/s) - h \rfloor$ as claimed.

Before computing the time complexity, let us observe that every subgraph H of $\tilde{G}$ has $O(s|V(H)|)$ edges. Indeed, it is known [22] that every s-separable graph has treewidth at most $4s$. Moreover, treewidth-k n-vertex graphs have no more than kn edges, due to the simplicial elimination scheme of their minimal triangulation. It follows that H has at most $4s|V(H)|$ edges.

Let us show now that it takes a linear time to compute the biseparator B, the root of T. At each iteration of the main loop, a separator of H is computed in $O(|V(H)| + |E(H)|)$ time by assumption. The updates of V_1, V_2, and H are also determined in $O(|V(H)| + |E(H)|)$. We have seen that $|E(H)| \leqslant 4s|V(H)|$. At the i-th iteration of the main loop, $|V(H)| \leqslant n/2^i$. So, the i-th iteration takes $O(|V(H)| + |E(H)|) = O(sn/2^i)$ time. The time for computing the biseparator is therefore $\sum_{i=0}^{\log(n/s)} O(sn/2^j) \leqslant c \cdot sn$ for some constant $c > 0$.

In total, the bidecomposition T is computed recursively in time $t(n) \leqslant c \cdot sn + 2t(n/2)$ if $n > s$, and $t(n) = O(1)$ in if $n \leqslant s$, which is $t(n) = O(sn \log(n/s))$. □

2.3 The Labels

Let us fix a (k, s)-graph G with n vertices. So G has a spanning (k, s)-triangulation $\tilde{G}$ that is s-separable. Let T be a bidecomposition for $\tilde{G}$ satisfying Lemma 1. For every vertex u of G, let K_u be a complex rooted at u obtained by adding to u every incident edge leading to a neighbors of u in G that is an out-neighbor in $\tilde{G}$. By construction, $V(K_u)$ induces a clique of at most $k + 1$ vertices in $\tilde{G}$. Hence, the parts of each vertex of K_u are pairwise related in T.

We define the function $\beta(h) = \sum_{i=0}^{h} s \lfloor \log(2n/s) - i \rfloor$. Intuitively, $\beta(h)$ represents the maximum number of vertices of G contained in the parts of a branch[8] of T of length $h + 1$. We have also $\beta(h) = s(h + 1)(\lfloor \log(2n/s) \rfloor - h/2)$.

Let X be a part of T at depth h. Te root of T is at depth $h = 0$. By Lemma 1, $|X| \leqslant \beta(h) - \beta(h - 1)$. We denote by $\mathrm{path}(X)$ the binary word defining the unique path from the root of T to X. The length of $\mathrm{path}(X)$ is $|\mathrm{path}(X)| = h$. We associated with each $u \in X$, its *rank*, a unique integer $\mathrm{rank}(u) \in [0, |X|)$, and its *position*, defined by $\mathrm{pos}(u) = \mathrm{rank}(u) + \beta(h - 1)$. The *apex* of u is the vertex a_u of K_u with maximum position, i.e., such that $\mathrm{pos}(a_u) = \max_{v \in V(K_u)} \mathrm{pos}(v)$.

Observe that the positions are relative to a branch of T: every pair of vertices whose parts are on the same branch have distinct positions, and thus the parts of any two vertices having the same positions cannot be related.

Let u be a vertex of G, and let a_u be the apex of K_u in T. The label of vertex u is defined by the following quadruple:

$$\mathrm{label}(u) = (\mathrm{path}(T_{a_u}), \mathrm{rank}(a_u), P_u, r_u)$$

where $P_u = \{\mathrm{pos}(v) \mid v \in V(K_u), v \neq a_u\}$ and $r_u = |\{p \in P_u \mid p < \mathrm{pos}(u)\}|$. Roughly speaking, P_u is the set of positions of all the vertices of K_u but its apex, and r_u is the rank of the root of K_u in P_u. So $r_u = 0$ if $\mathrm{pos}(u)$ is the smallest position in P_u; $r_u = |P_u|$ if $u = a_u$ is the apex itself.

Let $\mathrm{pos}(K_u) = \{\mathrm{pos}(v) \mid v \in V(K_u)\}$. It is not difficult to see that the complex K_u is uniquely defined by the pair $(\mathrm{pos}(K_u), \mathrm{path}(T_{a_u}))$, i.e., the set of its positions and the path leading to its apex. Indeed, as said previously, vertices lying on the same branch of T have pairwise distinct positions, and vertices lying on different branches can be identified from the path of their apices (that must therefore differ). The set $\mathrm{pos}(K_u)$ is not a field of $\mathrm{label}(u)$, P_u misses $\mathrm{pos}(a_u)$. However,

[8] A path that leads to the root of T.

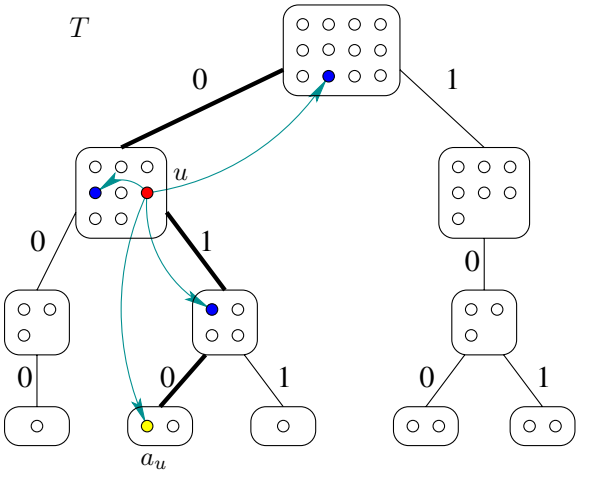


Fig. 2. A bidecomposition with a complex K_u of root u and apex a_u

it can be computed since $\text{pos}(a_u) = \text{rank}(a_u) + \beta(|\text{path}(T_{a_u})|)$. It follows that $\text{label}(u)$ is a (one-to-one) coding of K_u. In particular, $\text{label}(u) \neq \text{label}(v)$ since $K_u \neq K_v$ for distinct vertices $u \neq v$.

Lemma 2. *The labels are of* $\log n + 2k \log\log(n/s) + O(k \log(s/k))$ *bits.*

Proof. We assume that n, k, s are given. Let $w = \lfloor \log(2n/s) \rfloor$, and let $h = |\text{path}(T_{a_u})|$. The binary word $\text{path}(T_{a_u})$ is of length exactly h, and $\text{rank}(a_u) \in [0, \beta(h) - \beta(h-1))$. We have $\beta(h) - \beta(h-1) = s \lfloor \log(2n/s) - h \rfloor = s(w-h)$.

We write $\text{rank}(a_u) = x \cdot (w-h) + y$ where $x \in [0, s)$ and $y \in [0, w-h)$: $x = \lfloor \text{rank}(a_u)/(w-h) \rfloor$ and $y = \text{rank}(a_u) \bmod (w-h)$. We represent the two first fields $\text{path}(T_{a_u}), \text{rank}(a_u)$ by the binary string[9]:

$$S \;=\; 0^y \circ 1 \circ \sigma_x \circ \text{path}(T_{a_u})$$

where σ_x denotes the standard binary representation of x on $\lceil \log s \rceil$ bits. Given S (but ignoring h), one can extract, y and σ_x, thus $\text{path}(T_{a_u})$, h, and $\text{rank}(a_u) = x \cdot (w-h) + y$. The length of S is $|S| \leqslant (w-h-1)+1+\lceil \log s \rceil + h = w + \lceil \log s \rceil = \lfloor \log(2n/s) \rfloor + \lceil \log s \rceil \leqslant \log n + O(1)$. All the positions in P_u range in $[0, \beta(h))$. It follows that there are at most $\binom{\beta(h)}{|P_u|} \leqslant \binom{\beta(h)}{k}$ possible ways to select the set P_u having at most k positions. This can be coded with $\log \binom{\beta(h)}{k} + O(1)$ bits. Finally, $r_u \in [0, |P_u|]$, thus there are $k+1$ possible values. This can be coded with $\lceil \log(k+1) \rceil$ bits. In total, the length of $\text{label}(u)$ is at most:

$$|\text{label}(u)| \;\leqslant\; \log n + \log \binom{\beta(h)}{k} + O(\log k) \tag{1}$$

[9] We denote by $\circ$ the word concatenation.

By Lemma 1, the size of the parts in T of depth $\log(n/s)$ are of size at most $s \cdot (\log(2n/s) - \log(n/s)) = s$. From the while-condition in Algorithm BISEPARATOR, if $|V(H)| \leqslant s$, then the input graph is not separated at all, implying that the part is actually a leaf of T. Therefore $h \leqslant \log(n/s)$.

We have $\beta(h) \leqslant \beta(\log(n/s)) \leqslant (\log(n/s)+1) \cdot s \cdot \log(2n/s)) = s \cdot \log^2(2n/s)$. Thus $\log \binom{\beta(h)}{k} \leqslant k \log(\beta(h)/k) + O(k) \leqslant k \log(s/k \cdot \log^2(2n/s)) + O(k) \leqslant 2k \log\log(n/s) + O(k \log(s/k))$. Plugging this latter bound in Eq. (1), we get the desired result. □

So, for treewidth-k graphs, that are $(k, k+1)$-graphs, we obtain labels of length roughly $\log n + 2k \log\log(n/k)$. For $(1,1)$-graphs (forests), this is $\log n + 2\log\log n$. Actually, a finer analysis shows that for trees the label length is no more than $\log n + 2\log\log n + 2$ for every $n \geqslant 5$, a competitive bound with the $\log n + 4\log\log n$ scheme of [3].

We finally observe that our scheme support parent and sibling queries in rooted trees since we associate with each node a coding of itself and its parent. The label length is $\log n + O(\log\log n)$ which is not optimal for parent but optimal for sibling queries due to the lower bound of $\log n + \Omega(\log\log n)$ of [2].

2.4 Adjacency Test

Let u and v be two vertices of G. We denote by $M = \beta(h)$ where h is the length of the longest common prefix between $\text{path}(T_{a_u})$ and $\text{path}(T_{a_v})$. The proof of the next result appears in the full version.

Lemma 3. *The vertices u and v are adjacent if and only if* $\text{label}(u) \neq \text{label}(v)$ *and if either* $\text{pos}(u) \in \text{pos}(K_v) \cap [0, M)$ *or* $\text{pos}(v) \in \text{pos}(K_u) \cap [0, M)$.

We now briefly explain how to implement the adjacency test given by Lemma 3 to perform in constant time under the standard $\Omega(\log n)$-bit word RAM computer model. First we observe that $\beta(h)$ can be computed in constant time using its closed formula: $\beta(h) = s(h+1)(\lfloor s\log(2n/s)\rfloor - h/2)$. Thus M can be computed in constant time, because the length of the common prefix between two $O(\log n)$-bit words can be computed using constant number of MSB[10], binary masks, and shifting operations. Now, to test if $\text{pos}(v) \in \text{pos}(K_u)$, we can test if $\text{pos}(v) = \text{pos}(a_u)$ or if $\text{pos}(v) \in P_u$. The position of the apex is $\text{pos}(a_u) = \text{rank}(a_u) + \beta(|\text{path}(T_{a_u})|)$. The position of u can be also determined from $\text{label}(u)$, since $\text{pos}(u)$ is $\text{pos}(a_u)$ if $r_u = |P_u|$, or the r_u-th element of P_u. It follows that testing if $\text{label}(u) \neq \text{label}(v)$ can be done by checking that $(\text{pos}(u), \text{path}(T_{a_u})) \neq (\text{pos}(v), \text{path}(T_{a_v}))$. Finally, computing $\text{pos}(u)$ and testing whether $\text{pos}(u) \in P_v$ rely on membership and select query in an integer subset $P_u \subset \{0, \ldots, \beta(h)\}$.

Using a compact static dictionary, one can implement such queries in constant time using a data-structure of $(1+o(1))\log\left(\binom{\beta(h)}{|P_u|}\right)$ bits [21]. We have already seen in the proof of Lemma 2 that $\log \binom{\beta(h)}{k} \leqslant 2k\log\log(n/s) + O(k\log(s/k))$.

[10] Most Significant Bit, or integer LOG function.

Lemma 4. *The family of n-vertex (k,s)-graphs enjoys an adjacency labeling scheme with label of $\log n + (2k + o(1)) \log\log(n/s)$ bits, and with a constant adjacency query time.*

We prove Theorem 1 by plugging $s = k+1$ and combining all lemmas, and using the fact that a $(k, k+1)$-triangulation for treewidth-k graphs can be computed in linear time for fixed k [6].

3 Concluding Remarks and Open Problems

We have proposed a new implicit representation for planar graphs, and more generally to graphs excluding a fixed minor, with asymptotically $2\log n$ bits per vertex. Improving this bound would require a totally different approach, since decomposing into two or more subgraphs inevitably leads to a $c\log n$ representation with $c \geqslant 2$.

From the lower bound side, the number $\psi(n, H)$ of n-vertex labeled H-minor-free graphs is $n! \cdot 2^{hn+o(n)}$, where h depends only on H [20]. Therefore, the trivial information-theoretic lower bound for adjacency is $\frac{1}{n}\log\psi(n, H) \sim \log n + h$ bits for at least one label. This leads to the natural question we propose as open problem:

Conjecture. The family of H-minor-free graphs supports an adjacency labeling scheme with $\log n + h$ bit labels where $h = h(H)$.

Proving a labeling scheme of $\log n + O(k)$ for treewidth-k graphs would be already very interesting, since not only if would match our lower bound of $\log n + \Omega(k)$, but also prove an optimal bound for trees (up to an additive constant) which is still open.

References

1. Abiteboul, S., Alstrup, S., Kaplan, H., Milo, T., Rauhe, T.: Compact labeling schemes for ancestor queries. SIAM J. on Computing 35, 1295 (2006)
2. Alstrup, S., Bille, P., Rauhe, T.: Labeling schemes for small distances in trees. SIAM J. on Discrete Mathematics 19, 448–462 (2005)
3. Alstrup, S., Rauhe, T.: Small induced-universal graphs and compact implicit graph representations. In: 43^{rd} Annual IEEE Symp. on Foundations of Computer Science (FOCS), nov, pp. 53–62. IEEE Computer Society Press, Los Alamitos (2002)
4. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. on Algeb. and Disc. Meth. 8, 277–284 (1987)
5. Bazzaro, F., Gavoille, C.: Localized and compact data-structure for comparability graphs. In: Deng, X., Du, D.-Z. (eds.) ISAAC 2005. LNCS, vol. 3827, pp. 1122–1131. Springer, Heidelberg (2005)
6. Bodlaender, H.L.: A linear time algorithm for finding tree-decompositions of small treewidth. SIAM J. on Computing 25(6), 1305–1317 (1996)
7. Bonichon, N., Gavoille, C., Labourel, A.: Short labels by traversal and jumping. In: Flocchini, P., Gkasieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 143–156. Springer, Heidelberg (2006)

8. Breuer, M.A.: Coding the vertexes of a graph. IEEE Transactions on Information Theory IT-12, 148–153 (1966)
9. Chung, F.R.K.: Universal graphs and induced-universal graphs. J. of Graph Theory 14, 443–454 (1990)
10. DeVos, M., Ding, G., Oporowski, B., Sanders, D.P., Reed, B., Seymour, P.D., Vertigan, D.: Excluding any graph as a minor allows a low tree-width 2-coloring. J. of Combinatorial Theory, Series B 91(1), 25–41 (2004)
11. Elmallah, E.S., Colbourn, C.J.: Partitioning the edges of a planar graph into two partial k-trees. Congressus Numerantium 66, 69–80 (1988)
12. Erdös, P., Gerencsér, L., Máté, A.: Problems of graph theory concerning optimal design. In: Colloq. Math. Soc. Janos Bolyai 4: Combinatorial theory and its applications, vol. 1, pp. 317–325. North-Holland, Amsterdam (1970)
13. Fraigniaud, P., Gavoille, C.: Routing in trees. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 757–772. Springer, Heidelberg (2001)
14. Gavoille, C., Paul, C.: Optimal distance labeling schemes for interval and circular-arc graphs. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 254–265. Springer, Heidelberg (2003)
15. Gilbert, J.R., Rose, D.J., Edenbrandt, A.: A separator theorem for chordal graphs. SIAM J. on Algebraic and Discrete Methods 5, 306–313 (1984)
16. Gonçalves, D.: Edge partition of planar graphs into two outerplanar graphs. In: 37^{th} Annual ACM Symp. on Theory of Computing (STOC), pp. 504–512. ACM Press, New York (2005)
17. Étude de différents problèmes de partition de graphes, PhD thesis, Université Bordeaux 1, Talence, France, Dec (2006)
18. Kannan, S., Naor, M., Rudich, S.: Implicit representation of graphs. In: 20^{th} Annual ACM Symp. on Theory of Computing (STOC), pp. 334–343. ACM Press, New York (1988)
19. Korman, A., Peleg, D.: Compact separator decompositions and routing in dynamics trees. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596. Springer, Heidelberg (2007)
20. Norine, S., Robertson, N., Thomas, R., Wollan, P.: Proper minor-closed families are small. J. of Combinatorial Theory, Series B 96(5), 754–757 (2006)
21. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM J. on Computing 31(2), 353–363 (2001)
22. Reed, B.: Finding approximate separators and computing treewidth quickly. In: 24^{th} Annual ACM Symp. on Theory of Comp (STOC), pp. 221–228. ACM Press, New York (1992)
23. Robertson, N., Seymour, P.D.: Graph minors. III. planar tree-width. J. of Combinatorial Theory, Series B 36, 49–64 (1984)
24. Schnyder, W.: Embedding planar graphs on the grid. In: 1^{st} Symp. on Discrete Algorithms (SODA), pp. 138–148. ACM Press, New York (1990)
25. Thorup, M., Zwick, U.: Compact routing schemes. In: 13^{th} Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA), pp. 1–10. ACM Press, New York (2001)

Dynamic Plane Transitive Closure

Krzysztof Diks[1] and Piotr Sankowski[1,2]

[1] Institute of Informatics, Warsaw University, Warsaw, Poland
[2] Dipartimento di Informatica e Sistemistica,
University of Rome "La Sapienza", Rome, Italy

Abstract. In this paper we study the problem of transitive closure in dynamic directed plane graphs. We show a dynamic algorithm supporting updates and queries in worst-case $\tilde{O}(\sqrt{n})$ time[1]. This is the first known algorithm for this problem with almost linear update time and query time product.

1 Introduction

The transitive closure problem is one of the most fundamental graph problems. The question if there is a path in the graph from one vertex to another is the simplest question one can ask. Although it has been extensively studied there are still many open problems left and the new one emerge, e.g., the complexity of the problem for sparse graphs seems to be not fully understood [18]. Moreover, even more interesting questions arise in the case of the dynamic transitive closure problem, which has attracted a lot of attention in recent years, e.g., the fully dynamic algorithms for general graphs have been presented in the following papers [4,6,8,7,1,13,12,14,15], whereas the only fully dynamic algorithm for the planar graphs has been presented in [16]. The algorithm of Subramanian [16] is based on sparse certificates for reachability information in planar graphs and supports updates and queries in $\tilde{O}(n^{\frac{2}{3}})$ amortized time. It should be noted that this algorithm was obtained before the presentation of any of the dynamic algorithm for general graphs. Furthermore, the fastest known algorithm for general graphs that supports updates and queries in $O(n^{1.5})$ worst-case time [15] works much slower than the algorithms developed for planar graphs, whereas the fastest algorithm for sparse graphs works in $O(n)$ query and $O(n \log n)$ amortized update time [14]. Moreover, one should note that a naive graph search gives an $O(1)$ update and $O(n)$ query time algorithm, which beats both of these general graph approaches.

Here we improve over the result of Subramanian [16] by showing the algorithm supporting updates and queries in $\tilde{O}(n^{\frac{1}{2}})$ worst-case time, but for a more restricted class of plane graphs, i.e., we assume that the embedding of the graph during edge updates cannot change. This is a rather weak assumption because in possible applications the embedding of the planar graph should fixed because

[1] $\tilde{O}$ denotes the so-called "soft O" notation, i.e. $f(n) = \tilde{O}(g(n))$ iff $f(n) = O(g(n) \log^k n)$ for some constant k.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 594–604, 2007.

the nodes might correspond to existing locations. From the other side it is known that the reachability information in much more restricted class of graphs, i.e., plane directed acyclic graphs (DAGs) with one source and one sink, can be maintained in $\tilde{O}(1)$ time [5].

Our algorithm is based on sparse reachability oracles, i.e., data structures that can answer reachability queries fast and in the same time have small size. The first example of such a data structure was given by Subramanian [16] who has shown that reachability information about k vertices located on a constant number of faces can be encoded into a graph of size $O(k \log k)$. This idea has been ingeniously extended by Thorup [17] who has shown that all the reachability information in a planar graph can be represented by a data structure of size $O(n \log n)$ that can answer queries in constant time. The main idea of both algorithms is separating directed path technique (see Lemma 1). We simplify slightly the construction of the sparse certificates and use some previously not used properties of the planar graphs, i.e., Monge property for the paths. This allows us to show how to compute the union of two sparse reachability certificates in time proportional to their size. This operation is essential to combine the reachability certificates with the separator techniques [10,11] in dynamic setting. Using both techniques we are able to present a data structure that support updates and queries in $\tilde{O}(n^{\frac{1}{2}})$ worst-case time.

The data structures of Subramanian [16,9] and Thorup [17] can be adopted to maintain information on approximate distances in planar graphs as well. In our case it is also possible to use the same extension and obtain a dynamic algorithm for maintaining approximate distances in planar graph. The algorithm supports update and query operations in $\tilde{O}(\epsilon^{-1}\sqrt{n})$ worst-case time. However, due to the space limitations the details of this algorithms will be included in the full version of the paper.

The paper is organized as follows. In the next section we introduce the notion of the sparse certificates and show how they can be constructed. In Section 2 we show how to compute the union of two sparse certificates. In the next section we present the decomposition technique based on separators. The decomposition together with the union operation is the main of the dynamic algorithm for transitive closure presented in Section 5. The final conclusions and open problems are contained in Section 6.

2 Sparse Reachability Certificates

In this section we show how to construct small certificates for the reachability information in planar graphs, which is inspired by the construction from [16] and [17]. Let $G = (V, E)$ be a directed planar graph and let $S \subseteq V$ be a set of k vertices lying on a boundary of face f. We would like to construct data structure of size $\tilde{O}(|S|)$ that would allow us to answer queries about the reachability only between the vertices of S in $\tilde{O}(1)$ time. The main idea of the path-detour technique uses the next lemma in order to construct a set of separating paths. In the next section we show how to represent the reachability information when

the vertices in S lay on a constant number of faces. In Section 5 we present how this data structure can be used to represent all the reachability information.

Lemma 1 (Subramanian '93). *One can construct in $O(|V| + |E|)$ time two paths π_1, π_2 in G such that the set S is divided in $G - \pi_1 - \pi_2$ into two parts S_1 and S_2 neither having more than $\frac{3}{4}$'ths of the nodes of S and there exist sets $Q_1 \subseteq S_1$ and $Q_2 \subseteq S_2$ such that there is no path between $S_1 - Q_1$ and $S_2 - Q_2$, and such that:*

- *every path between Q_1 and S_2 intersects the path π_1,*
- *every path between Q_2 and S_1 intersects the path π_2.*

Proof. Consider the following procedure:

- We number the nodes clockwise along the face f and divide them into four equal continuous subsets A_1, A_2, A_3 and A_4.
- If there is a path p from A_1 to A_3 or from A_2 to A_4 we use it to divide S into S_1 and S_2, and we set $\pi_1 = p$, $\pi_2 = \emptyset$, $Q_1 = S_1$ and $Q_2 = \emptyset$.
- We set a_i, for $i = 1, 2$, to be the lowest numbered node in A_{2i-1} such that there exists a path from or to some node b_i in A_{2i} and we set π_i to be this path. If there is no such path we set $\pi_i = \emptyset$.
- The set S_1 includes the nodes from a_1 to a_2 not including a_2, the set Q_1 includes the nodes of S_1 starting from b_1, whereas the set S_2 includes the nodes a_2 to a_1 not including a_1 and Q_2 includes the nodes of S_2 starting from b_2.

First of all note that there is no path between $S_1 - Q_1$ and $S_2 - Q_2$ because either $S_1 - Q_1$ is empty or there is no path from A_1 to A_3 or from A_2 to A_4. Additionally, it follows from the constructions that the paths between Q_1 and S_2 have to intersect the path π_1 and similarly the paths between Q_2 and S_1 have to intersect the path π_2.

In order to find the nodes a_i one has to collapse the nodes of A_{2i} into a single super node s and perform a DFS on the normal and reversed graph. In such a way we find all nodes from A_{2i-1} that have a path to or from A_2 and we just need to chose the one with the smallest number. This takes $O(|V| + |E|)$ time and the lemma follows.

We will now show that all the paths intersecting a directed path can be sparsely represented. The following lemma was proven by Subramanian [16]. However, here we give different proof that includes the construction of our extended oracle.

Lemma 2. *Let S_1 and S_2 be defined as in Lemma 1, then there exists a data structure of size $O(k)$ that can answer queries about the reachability from S_1 to S_2 in constant time can be constructed in $O(|E| + |V|)$ time.*

Proof. Let us start by representing the information about the paths from Q_1 to S_2. For the path π_1 we maintain an ordered list L_{π_1} of the nodes in $Q_1 \cup S_2$. Now for each $x \in Q_1$ we find the first vertex x_{π_1} on π_1 that is reachable from x,

whereas for each $x \in S_2$ we find the last vertex x_{π_1} on π_1 from which x can be reached. The order of the vertices in L is determined by the order of vertices x_{π_1} on π_1. Let us denote by $\mathrm{ord}_{L_{\pi_1}}(x)$ the position of x in L_{π_1}. Note now that $y \in S_2$ is reachable from $x \in Q_1$ iff $\mathrm{ord}_{L_{\pi_1}}(x) \leq \mathrm{ord}_{L_{\pi_1}}(y)$. In order to determine the vertices x_{π_1} for the set Q_1 we simply run the DFS procedure for reversed edges starting at the vertices on π_q in their order. In this way we can determine for each $s \in \pi_1$ all the vertices $x \in Q_1$ such that $x_{\pi_1} = s$. The vertices x_{π_1} for the set S_2 can be determined in the same way by considering the graph with normal direction of edges. In order to describe the paths from Q_2 to S_1 we construct the list L_{π_2} in the same way.

In the Subramanians data structure the sparse certificates are stored as graphs, i.e., for each vertex x he adds the edge (x, x_{π_1}) to the sparse certificate. However, note that the ordered lists L_{π_1}, L_{π_2} determine the reachability from S_1 to S_2 even without the knowledge of the paths π_1 and π_2. This allows us for the following observation.

Corollary 1. *If there exist paths π_1, π_2, sets S_1, S_2, Q_1 and Q_2 satisfying Lemma 1, than there exist lists L_{π_1}, L_{π_2} such that:*

- *there exists a path from $x \in Q_1$ to $y \in S_2$ iff $\mathrm{ord}_{L_{\pi_1}}(x) \leq \mathrm{ord}_{L_{\pi_1}}(y)$,*
- *there exists a path from $x \in Q_2$ to $y \in S_1$ iff $\mathrm{ord}_{L_{\pi_2}}(x) \leq \mathrm{ord}_{L_{\pi_2}}(y)$,*

where the inequality is false if x or y is not included in the list.

Lemma 3 (Subramanian '93). *Let S_1 and S_2 be defined as in Lemma 1 and let V_i be the set of the vertices of the graph G such that $x \in V_i$ if and only if there is a directed path in $G - \pi_1 - \pi_2$ from $u \in S_i$ to $v \in S_i$ passing through x, then the sets V_i can be constructed in $O(|V| + |E|)$ time and $V_1 \cap V_2 = \emptyset$.*

Proof. Let $x \in V_1 \cap V_2$ then from definition of V_i there would be a path from $u \in S_1$ to $v \in S_2$ passing through x, that does not intersect π_1 and π_2. This contradicts Lemma 1.

Now in order to determine V_i we combine the nodes in S_i into a single node s and find the set of vertices in $G - \pi_1 - \pi_2$ that can be reached from s and can reach s.

Theorem 1. *Let $G = (V, E)$ be a directed planar graph and let S be a set of nodes lying on the boundary of a face f. In $O(n \log n)$ time we can construct a data structure X of size $O(k \log k)$ that can answer the reliability queries about the vertices in S in $O(\log k)$ time.*

Proof. The data structure X can be constructed in the following way:

1. we find two paths π_1, π_2 in G as given in Lemma 1,
2. we sparsify the information about the reachibility between S_1 and S_2 using the path π_1 and π_2 by applying Lemma 2 twice,
3. we divide the vertices of the graph G into two parts V_1 and V_2 using Lemma 3,

4. we construct subgraphs G_1, G_2 induced by the set of vertices S_1, S_2 and compute recursively the reachability certificates in G_1 and G_2,
5. finally we union the three certificates to get the final certificate for S in G.

It is easy to see that the recursive calls take $O(n \log n)$ time and that the constructed size of the data structure is $O(k \log k)$. In the obtained data structure for each vertex in $x \in S$ we maintain the position of x on $O(\log k)$ lists. In order to answer the query if there is a path from x to y, we need to check whether there exists a list on which x precedes y and this takes $O(\log k)$ time.

The data structure satisfying the above theorem is called a *sparse certificate*. We can easily transform the sparse certificate X into a graph G_X that represents all the information stored in X. However, the obtained graph G_X is not planar.

Corollary 2. *The data structure X from Theorem 1 can be transformed into a graph $G_X = (V_X, E_X)$, $S \subseteq V_X$, $|E_X| = O(|S| \log |S|)$ representing correctly reachability information in X in $|X|$ time.*

Proof. We simply transform each sorted list into directed paths and add edges from the vertices of S to the corresponding vertices on the paths.

3 Union of Sparse Certificates

Now we know how the reachability information in planar graphs can be presented and we are ready to show how a union of two certificates can be efficiently computed. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be a division of $G = (V, E)$ in to two subgraphs, i.e., $V_1 \cup V_2 = V$, $E_1 \cup E_2 = E$ and $E_1 \cap E_2 = \emptyset$. Let P_i be the set of the nodes lying on the outer face f_i of G_i. Let us assume that $P_1 \cup P_2 \neq \emptyset$ and let S be the set of nodes lying on the outer face of $G_1 \cup G_2$. Of course we have $S \subseteq P_1 \cup P_2$. Now let us assume that we are given sparse certificates X_1 and X_2 for P_1 in G_1 and respectively P_2 in G_2. Now we want to compute the certificate X for S in G.

Lemma 4. *One can construct the sets S_1, S_2, Q_1, Q_2 such there exist paths π_1, π_2 satisfying Lemma 1 in $O(|X_1| + |X_2|)$ time.*

Proof. Note that in the proof of Lemma 1 we only need to run several DFS procedures. Instead of running them on the graph G we can run them on the union of the graph G_{X_1} and G_{X_2} obtained from Corollary 2. In such a way we get the sets S_1, S_2, Q_1 and Q_2, but the paths we get come from the sparse certificates. Hence we cannot construct them explicitly as it might take too much time, but we know that they exist.

In order to construct sparse certificates we use recursion and divide the sparse certificates into smaller pieces. The algorithm is included in the proof of the following lemma.

Lemma 5. *A data structure of size $O(|S|)$ that answers queries about the reachability from Q_1 to S_2 in constant time can be constructed in $\tilde{O}(|X_1| + |X_2|)$ time.*

Proof. Our goal is to construct a list L_{π_1} as given in the proof of Lemma 2. However, now we know only that the path π_1 exists and we know its direction. Let us assume that it is going upwards. We can construct the list using recursion in the following way:

1. we choose a vertex v dividing Q_1 into two continuous sets Q_1' and Q_1'' of equal size — v is not included in Q_1' nor Q_1'' and Q_1' lays below Q_1''.
2. let us find the lowest vertex $w \in S_2$ reachable from v and denote the path from v to w by ρ,
3. divide S_2 using w into two continuous sets S_2' and S_2'' — w is not included in S_2' nor S_2'' and S_2' lays below S_2'',
4. divide X_1 into X_1' and X_1'' along the path ρ,
5. divide X_2 into X_2' and X_2'' along the path ρ,
6. recursively find list L'_{π_1} for the sets Q_1' and S_2' using X_1' and X_2',
7. recursively find list L''_{π_1} for the sets Q_1'' and S_2'' using X_1'' and X_2'',
8. set $L_{\pi_1} = L'_{\pi_1} \cdot (v, w) \cdot L''_{\pi_1}$, where $\cdot$ is the concatenation operation.

Let us start by explaining how the certificates X_i are divided along the path ρ. We follow the path ρ in G_X and whenever it crosses a list L_π we split L_π into two disjoint parts. The one below ρ is included into X_i' and the one above ρ is included into X_i''. Note that $X_1' \cup X_2'$ correctly describes all the paths from Q_1' to S_2' because if any of such paths crosses ρ it must cross it an even number of times, so instead of going with this path we can shortcut with the path ρ. The paths not going from S_1' to S_2' may not be correctly described but we do not use them in the recursion.

The correctness of the algorithm follows from the observation that any path from Q_1' to S_2'' has to cross one of the paths π_1 or ρ before their intersection. Hence all vertices in Q_1' can reach exactly the same vertices in S_2'' as v does. Furthermore, there is no path from Q_1'' to S_2', because if there were any it would cross the path ρ, and the vertex w would not be the lowest one. Knowing this we can place the list L'_{π_1} before the elements (v, w) and the list L''_{π_1}.

Now let us look on the running time of the above algorithm. Note that there are at most $\log |Q_1|$ levels of recursion, because each time Q_1 is divided into two equal size sets. Moreover, in each recursion level the total work is bounded by the size of X_1 and X_2 at the beginning, because X_1 and X_2 are split into disjoint parts so the total size of the pieces on each level cannot increase. Hence the total work in the recursion is bounded by $O(\log |Q_1|(|X_1| + |X_2|)) = \tilde{O}(|X_1| + |X_2|)$.

Having the above lemma we are ready to show how an union of two sparse certificates can be computed.

Theorem 2. *Let G_1 and G_2 be a division of G in to two subgraphs. Let P_i be the sets of the nodes lying on the outer face f_i of G_i. Let us assume that $P_1 \cup P_2 \neq \emptyset$ and let S be the set of nodes lying on the outer face of $G_1 \cup G_2$. Let X_1 and X_2 be sparse certificates for P_1 in G_1 and respectively for P_2 in G_2. The sparse certificate X for S in G can be constructed in $\tilde{O}(|X_1| + |X_2|)$ time.*

Proof. The sparse certificate X can be constructed in the following way:

1. we find the sets S_1, S_2, Q_1 and Q_2 as given in Lemma 4,
2. we sparsify the information about the reachability between S_1 and S_2 using the path π_1 and π_2 and applying twice the Lemma 5,
3. we divide the sparse certificate X_1 into $X_{1,1}$ and $X_{1,2}$ using the sets S_1 and S_2,
4. we divide the sparse certificate X_2 into $X_{2,1}$ and $X_{2,2}$ using the sets S_1 and S_2,
5. we union recursively the reachability certificates $X_{1,1}$ and $X_{2,1}$ in order to obtain the certificate for S_1,
6. we union recursively the reachability certificates $X_{1,2}$ and $X_{2,2}$ in order to obtain the certificate for S_2,
7. finally we union the three certificates to get the final certificate for S in G.

The division of the sparse certificate X_i into $X_{i,1}$ and $X_{i,2}$ using the sets S_1 and S_2 can be done by looking on every list L_π and splitting it according with the sets S_1 and S_2. It follows from Lemma 1 and Lemma 4 that there are at most $O(\log(|X_1|+|X_2|))$ levels of recursion. Whereas from Lemma 5 we get that each recursion level takes $\tilde{O}(|X_1|+|X_2|)$ time. Hence the theorem follows.

4 Planar Graph Decomposition

A *decomposition* of a graph $G = (V, E)$ is a set of subsets $P_1, \ldots, P_k$ such that $P_1 \cup \ldots \cup P_k = V$ and such that for $e \in E$ there is exactly one P_i containing e. A node $v \in P_i$ is a *border node* of P_i if there exists $w \in P_j$ for $i \neq j$, such that $(v, w) \in E$. A subgraph of G induced by P_i is called a *piece*. If a piece is decomposed once again the set of its border nodes is formed by the set of original border nodes and the border nodes introduced by the new decomposition. Given an embedding of a piece we say that a *hole* in it is a bounded face composed of border nodes. We assume that we are given the following recursive decomposition of the planar graph.

Theorem 3 (Fakcharoenphol and Rao [3]). *There exist a recursive decomposition where at each level a piece with n nodes and r border nodes is divided into two subpieces such that each subpiece has no more than $\frac{2}{3}n$ nodes and at most $\frac{2}{3}r + c\sqrt{n}$ border nodes, for some constant c. The decomposition stops when each piece contains one edge. Moreover, each piece contains at most a constant number h of holes.*

In the case of some interesting classes of graphs, e.g., grids graphs, we have $h = 0$ and the theorems from previous section can be applied directly. However when $h \neq 0$ we have to deal with the problem rather in a standard way, see [16,9,3]. For each of the h holes we apply Theorem 2 separately in order to describe the reachability information between the vertices of the hole. Whereas in order to describe the reachability information between different holes we have to modify slightly the data structure. It can be achieved by observing that for each pair

of holes one can find a set of three separating paths. Let us assume that we are dealing with vertices laying on face f and g. We find a path ρ going from f to g, if there is no such path we are done, because there is no path between f and g. Now we can union f, g and ρ and deal with them as with a single face in order to apply Lemma 4. In such a way we obtain a set of three separating paths instead of two. Using these paths we can construct a sparse certificate representing the reachability between f and g. Because there are $\frac{h(h-1)}{2}$ possible pairs of holes and because h is constant the running time of the algorithm increases only by a constant factor when we consider holes in the decomposition.

5 Dynamic Maintenance of Sparse Certificates

In this section we use the ideas presented in the previous section in order to construct dynamic algorithms. Here we assume that we are working with a plane dynamic graph, where the embedding of the graph does not change, i.e., we assume that the relative position of the vertices does not change. Our dynamic data structure is constructed in a very simple way. We take the recursive decomposition given by Theorem 3 and starting from the lowest level pieces we compute the sparse certificates for every piece by unioning sparse certificates for subpieces with use of Theorem 2. This takes all together $\tilde{O}(n)$ time. Now let us present how queries and updates can be realized.

Let us assume that we want to insert an edge (v, w). We can note that because the graph remains plane the edge can be inserted only inside an embedded face of the graph. Hence such an operation cannot increase the number of holes in any piece, e.g., by dividing a hole into two, because holes have to contain some other piece inside and never are faces in the original graph. The insertion of the edge changes only the reachability information for pieces it is complectly included. Hence we do as follows, for every piece P such that $v \in P$ but $w \notin P$ we add v as a border node and recompute the reachability information in P by unioning the certificates for the subpieces. Similarly we add w as a border node to the subpieces such that $w \in P$ but $v \notin P$. Now for every piece such that $v \in P$ and $w \in P$ we recompute the reachability information again by unioning the certificates for the subpieces. Moreover, after every $\sqrt{n}$ operations we recompute the structure from the scratch. When an edge (v, w) has to be removed it is enough to process every piece P such that $v \in P$ and $w \in P$ and recompute the reachability information for P by unioning the certificates for the subpieces.

Note that every insert operation can increase the number of border nodes by one in some piece. Hence after $\sqrt{n}$ operation a piece can have at most $O(\sqrt{n})$ border nodes, so each operation on it takes at most $\tilde{O}(\sqrt{n})$ time. Additionally, note that in the decomposition there are at most $O(\log n)$ levels so v and w can be contained in at most $O(\log n)$ pieces. Hence the insert and delete operation need $\tilde{O}(\sqrt{n})$ amortized time including the cost of the recomputation of the whole structure every $\sqrt{n}$ steps. This bound can be made worst-case by using the global rebuilding technique in which we keep two copies of the data structure. One is used for answering queries when the second one is being recomputed.

Let us assume that we want to check if w can be reached from v. Let us denote by $\mathcal{P}_v$ and $\mathcal{P}_w$ the sets of pieces of the decomposition that include v or w respectively. For every piece in $\mathcal{P}_v \cup \mathcal{P}_w$ we take a sparse certificate and build a graph representing it using Corollary 2. Next we union all of the obtained graphs in order to obtain a skeletal graph $G_{v,w}$.

Lemma 6. *There is a path in $G_{v,w}$ from v to w if and only if there is a path from v to w in G.*

Proof. First of all note that each vertex becomes a border node at some level of the decomposition. Hence v and w are the vertices in $G_{v,w}$. Now consider any path p from v to w in G. There must be a piece P in $\mathcal{P}_v$ that includes whole the path p. Moreover, the same piece is included in $\mathcal{P}_w$ as well. Now we can recursively divide P according to the given decomposition into subpieces and obtain the set of border nodes of the pieces in $\mathcal{P}_v \cup \mathcal{P}_u$ crossed by p. Note that according to Theorem 1 the reachability information between the border nodes is correctly represented. Hence if there is a path in G from v to w, then there is a corresponding path in $G_{v,w}$. Now let us assume that there is no path from v to w in G, then trivially there is no path in $G_{v,w}$, because $G_{v,w}$ contains only a subset of the paths contained in G.

The above lemma states that the queries can be correctly processed. Now let us consider the running time needed to answer a query. As argued before we know that each of the pieces can contain at most $O(\sqrt{n})$ border nodes and the vertices v and w can be contained in $O(\log n)$ pieces. Hence the size of the graph $G_{v,w}$ is $\tilde{O}(\sqrt{n})$ and the queries can be realized in $\tilde{O}(\sqrt{n})$ time. As a consequence we obtain the following theorem.

Theorem 4. *There exists an dynamic algorithm for directed plane graphs supporting updates and reachability queries in $\tilde{O}(\sqrt{n})$ worst-case time.*

Remark 1. The actual update time of the algorithm is $O(\sqrt{n}\log^3 n)$. Two $\log n$ factors are accumulated from Lemma 5 and Theorem 2, whereas the third factor is the result of the graph decomposition given above. The query time is $O(\sqrt{n}\log^2)$, because this is the size of the graph that represents reachability between two vertices.

6 Conclusions and Open Problems

We have presented a first algorithm for dynamic transitive closure in directed plane graphs supporting updates and queries in $\tilde{O}(\sqrt{n})$ worst case time. The algorithm can be as well used for maintaining ϵ approximate distances in undirected planar graphs in $\tilde{O}(\epsilon^{-1}\sqrt{n})$ worst-case time. However, the details of this algorithm due to space limitations are postponed to the full version of the paper. It seems that the presented result reaches the limits set by the separator techniques, i.e., in the current approach in each operation we need to process the separator of size $\Theta(\sqrt{n})$. However, the results for plane DAGs suggest that

breaking this limit might be possible. Nevertheless, the most interesting open problem is the question whether the technique can be generalized to planar graphs. Note that the information whether a dynamic graph is planar can be maintained in $\tilde{O}(\sqrt{n})$ time [2].

Acknowledgements

This work was partially supported by the EU within the 6th Framework Programme under contract no. 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS), by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP) under contract no. FP6-021235-2 ARRIVAL), and by the Polish Ministry of Science KBN grant N206 005 32/0807.

References

1. Demetrescu, C., Italiano, G.F.: Fully Dynamic Transitive Closure: Breaking Through the $O(n^2)$ Barrier. In: Proceedings of 41th annual IEEE Symposium on Foundations of Computer Science, pp. 381–389. IEEE Computer Society Press, Los Alamitos (2000)
2. Eppstein, D., Galil, Z., Italiano, G.F., Spencer, T.H.: Separator based sparsification for dynamic planar graph algorithms. In: STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pp. 208–217. ACM Press, New York, USA (1993)
3. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci. 72(5), 868–889 (2006)
4. Henzinger, M.R., King, V.: Fully Dynamic Biconnectivity and Transitive Closure. In: Proceedings 36th annual IEEE Symposiumon Foundations of Computer Science, pp. 664–672 (1995)
5. Husfeldt, T.: Fully dynamic transitive closure in plane dags with one source and one sink. In: European Symposium on Algorithms, pp. 199–212 (1995)
6. Khanna, S., Motwani, R., Wilson, R.H.: On Certificates and Lookahead on Dynamic Graph Problems. In: Proceedings 7th annual ACM-SIAM Symposiumon on Discrete Algorithms, pp. 222–231 (1996)
7. King, V.: Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In: Proceedings of 40th annual IEEE Symposium on Foundations of Computer Science, pp. 81–91. IEEE Computer Society Press, Los Alamitos (1999)
8. King, V., Sagert, G.: A Fully Dynamic Algorithm for Maintaining the Transitive Closure. In: Proceedings of the thirty-first annual ACM Symposium on Theory of Computing, pp. 492–498. ACM Press, New York (1999)
9. Klein, P.N., Subramanian, S.: A fully dynamic approximation scheme for all-pairs shortest paths in planar graphs. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1993. LNCS, vol. 709, pp. 442–451. Springer, Heidelberg (1993)
10. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM J. Applied Math., 177–189 (1979)
11. Miller, G.L.: Finding small simple cycle separators for 2-connected planar graphs. In: STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing, pp. 376–382. ACM Press, New York (1984)

12. Roditty, L.: A Faster and Simpler Fully Dynamic Transitive Closure. In: Proceedings of the fourteenth annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, pp. 404–412. ACM Press, New York (2003)
13. Roditty, L., Zwick, U.: Improved Dynamic Reachability Algorithms for Directed Graphs. In: Proceedings of the 43rd Symposium on Foundations of Computer Science, p. 679. IEEE Computer Society Press, Los Alamitos (2002)
14. Roditty, L., Zwick, U.: A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time. In: Proceeding of the 36th annual ACM Symposium on Theory of Computing, pp. 184–191. ACM Press, New York (2004)
15. Sankowski, P.: Dynamic transitive closure via dynamic matrix inverse. In: Proceedings of the 45th annual IEEE Symposium on Foundations of Computer Science, pp. 248–255. IEEE Computer Society Press, Los Alamitos (2004)
16. Subramanian, S.: A fully dynamic data structure for reachability in planar digraphs. In: Lengauer, T. (ed.) ESA 1993. LNCS, vol. 726, pp. 372–383. Springer, Heidelberg (1993)
17. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. J. ACM 51(6), 993–1024 (2004)
18. Yuster, R., Zwick, U.: Fast sparse matrix multiplication. ACM Trans. Algorithms 1(1), 2–13 (2005)

Small Stretch Spanners in the Streaming Model: New Algorithms and Experiments*

Giorgio Ausiello[1], Camil Demetrescu[1], Paolo G. Franciosa[2], Giuseppe F. Italiano[3], and Andrea Ribichini[1]

[1] Dipartimento di Informatica e Sistemistica, Sapienza Università di Roma, via Ariosto 25, I-00185 Roma, Italy
{ausiello,demetres,ribichini}@dis.uniroma1.it
[2] Dipartimento di Statistica, Probabilità e Statistiche Applicate, Sapienza Università di Roma, piazzale Aldo Moro 5, I-00185 Roma, Italy
paolo.franciosa@uniroma1.it
[3] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy
italiano@disp.uniroma2.it

Abstract. We present deterministic algorithms for computing small stretch spanners in the streaming model. An (α, β)-spanner of a graph G with n vertices is a subgraph $S \subseteq G$ such that for each pair of vertices the distance in S is at most α times the distance in G plus β. We assume that the graph is given as a stream of edges in arbitrary order, that the number of vertices and the number of edges are not known in advance and that only one pass over the data is allowed. In this model, we show how to compute a $(k, k-1)$-spanner of an unweighted undirected graph, for $k = 2, 3$, in $O(1)$ amortized processing time per edge/vertex. The computed $(k, k-1)$-spanners have $O(n^{1+1/k})$ edges and our algorithms use only $O(n^{1+1/k})$ words of memory space. In case only $\Theta(n)$ internal memory is available, our algorithms can be adapted to store some of the data structures in external memory. We complement our theoretical analysis with an experimental study that suggests that our approach can be of practical value.

1 Introduction

Graph spanners arise in many applications, including communication networks, computational biology, computational geometry, robotics and distributed computing ([3,5,10,11,13,14,15,18,19,20,21,22]). Given $\alpha \geq 1$ and $\beta \geq 0$, an (α, β)-spanner of a graph G is a subgraph S of G such that for each pair of vertices the distance in S is at most α times the distance in G plus β. A spanner with $\beta = 0$ is called *multiplicative*, while a spanner with $\alpha = 1$ is called *additive*.

* Partially supported by the Italian Ministry of University and Research under Project MAINSTREAM "Algorithms for Massive Information Structures and Data Streams".

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 605–617, 2007.

Baswana *et al.* [8] presented a deterministic $O(m+n)$ time algorithm to compute a $(k, k-1)$-spanner of size $O(k \cdot n^{1+1/k})$ of an unweighted graph with n vertices and m edges. In the same paper it is shown that any graph has a $(1,6)$-spanner having $O(n^{4/3})$ edges, which can be computed in $O(mn)$ worst-case time. For weighted graphs, a randomized $O(m+n)$ time algorithm that computes a multiplicative $(2k-1)$-spanner of size $O(k \cdot n^{1+1/k})$ has been given by Baswana and Sen [9]; a derandomization of this algorithm, still running in $O(m+n)$ worst case time, has been proposed by Roditty *et al.* [23].

In the dynamic setting, where edges may be added to or deleted from the original graph, few algorithms for updating multiplicative spanners have been proposed in the case of unweighted graphs. In [4], a 3-spanner and a 5-spanner are maintained under an intermixed sequence of $\Omega(n)$ edge insertions and deletions in $O(\Delta)$ amortized time per operation, where Δ is the maximum degree of the original graph. The maintained 3-spanner has $O(n^{3/2})$ edges, while the 5-spanner has $O(n^{4/3})$ edges. A faster randomized dynamic algorithm for general stretch factor k has been later given by Baswana [6]: a $(2k-1)$-spanner of expected size $O(k \cdot n^{1+1/k})$ can be maintained in $\widetilde{O}(\frac{m}{n^{1+1/k}})$ amortized expected time for each edge insertion/deletion. In the case of $k = 2, 3$ (3- and 5-spanners), the amortized expected time of the randomized algorithm becomes constant.

In the last years, some attention has been devoted to the computation of graph spanners in the streaming model. In this model, it is assumed that the input data is scanned as a sequence of values in arbitrary order and that the storage available is smaller than the input size. Thus, the algorithm is not allowed to randomly access the input. The complexity of a streaming algorithm is measured in terms of the number of passes over the input sequence (in the more restricted case, only one pass is allowed), the amount of internal storage used (always smaller than the input size) and the time needed to process the input.

In [17], Feigenbaum *et al.* presented a randomized algorithm for computing a multiplicative $(2k+1)$-spanner of an unweighted graph. If only one pass on the edges is allowed, the spanner has expected size $O(k \cdot n^{1+1/k})$, each edge is processed in expected $O(k^2 \cdot n^{1/k} \log n)$ time, using $O(k \cdot n^{1+1/k} \log n)$ memory words. Note that the size/stretch tradeoff of this algorithm is not optimal, since it always exists a $(2k-1)$-spanner with the same asymptotic size as the $(2k+1)$-spanner computed by [17]. This result has been recently improved by Baswana [7], who gives optimal (expected) size/stretch tradeoffs for multiplicative spanners in the streaming model: a $(2k-1)$-spanner of expected size $O(k \cdot n^{1+1/k})$ can be computed by a single pass over the edges in overall $O(m+n)$ time, using $O(k \cdot n^{1+1/k})$ memory space. However, both the algorithms in [7] and in [17], in order to apply successfully their randomization technique, need to know in advance the total number of vertices n in the graph. Similar results are given in [7] for computing spanners of weighted graphs in the *StreamSort* model of Aggarwal *et al.* [2], in which it is assumed that intermediate streams of data can be written, sorted and read again by the algorithm.

Our Results. In this paper we show that, in the more restricted streaming model in which only one pass over the data is allowed, it is possible to compute de-

terministically, for $k = 2, 3$, an optimal size $(k, k-1)$-spanner (i.e., containing $O(n^{1+1/k})$ edges in the worst case) in $O(1)$ amortized processing time per edge. Our algorithms use $O(n^{1+1/k})$ memory space in the worst case, and do not need to know in advance either the total number n of vertices or the total number m of edges of the graph. At any stage, the algorithms are able to compute on-line a $(k, k-1)$-spanner of the graph scanned so far, within the same time and space bounds. In case only $\Theta(n)$ internal memory is available, our algorithms can be also adapted to store some of the data structures in external memory. In the well known external memory model of [24], the same spanners can be computed using $O(n^{1+1/k}/B)$ external memory blocks, each of size B words. In this case, each edge is processed in $O(1)$ amortized time, plus $O(1/B)$ amortized block transfers. Our algorithms use simple data structures, are deterministic and rely on a relaxation of the deterministic clustering scheme introduced in [4].

We complement the theoretical analysis with a computational study in which our algorithm is compared to the randomized streaming algorithm for multiplicative $(2k-1)$-spanners by Baswana [7] and to an off-line algorithm by Zwick [25]. Our experiments show that streaming algorithms can be very efficient in practice, as the size and the stretch of the spanners produced are much smaller than the theoretical bounds. Moreover, our experimental findings suggest that small values of k seem to be the case of interest in practice, and that our algorithm tends to produce spanners of better quality than Baswana's algorithm, while still using a comparable amount of time and space resources.

2 Clustering and Induced Spanners

Let $G = (V, E)$ be an undirected graph, with V being the set of vertices and E the set of edges. We assume that the graph is given as a stream of edges and vertices (with edge (x, y) appearing after vertices x and y), that the number of vertices and edges are not known in advance and that only one pass over the data is allowed. We denote by m the current number of scanned edges and by n the current number of discovered vertices. The *distance* $dist_G(u, v)$ *from* u *to* v *in* G is given by the minimum length of a path in G from u to v (or $+\infty$ if there is no such path). Given $\alpha \geq 1$ and $\beta \geq 0$, an (α, β)-*spanner* of G is a graph $S = (V, E')$ with $E' \subseteq E$ such that for any $u, v \in V$, $\text{dist}_S(u, v) \leq \alpha \cdot \text{dist}_G(u, v) + \beta$. An $(\alpha, 0)$-spanner is called a *multiplicative spanner*, or more simply an α-spanner, while a $(1, \beta)$-spanner is called an *additive spanner*.

Our algorithms are based on a relaxation of the clustering schema introduced in [4]. We denote as the *neighborhood* of a vertex x the set $N(x) = \{x\} \cup \{y \mid (x, y) \in E\}$. Given a set of vertices $x_1, x_2, \ldots, x_k$, a *clustering* is a family of mutually disjoint sets $C(x_1)$, $C(x_2)$, ..., $C(x_k)$, with $C(x_i) \subseteq N(x_i)$ for $1 \leq i \leq k$. Each set $C(x_i)$ is called *cluster*, and x_i is referred to as its *center*. Note that the center x_i of a cluster $C(x_i)$ does not belong necessarily to $C(x_i)$, but may belong to a different cluster. A vertex is called *clustered* if it belongs to a cluster, and *free* otherwise; if y is clustered, $Cl(y)$ denotes the cluster containing y, while $center(y)$ denotes the center of $Cl(y)$. Starting from a clustering Γ of

a graph $G = (V, E)$, we define a *CC-subgraph on* Γ as the subgraph $S = (V, E')$, where E' as the union of the following edge sets:

- **cluster edges:** all edges (x, y) such that x is a center and $y \in C(x)$;
- **CC-bridge edges:** for each pair of clusters C_i, C_j, with $i \neq j$, one arbitrary edge $(x, y) \in E$ (if one exists) such that (x, y) is not a cluster edge, $x \in C_i$ and $y \in C_j$. We say that edge (x, y) *connects* clusters C_i and C_j;
- **free edges:** all edges $(x, y) \in E$ such that at either x or y is a free vertex.

The name CC-subgraph derives from the fact that bridges connect pairs of clusters. It is possible to prove that a CC-subgraph is a (3,2)-spanner of G.

By choosing a different set of bridge edges it is possible to obtain a (2,1)-spanner: we define a *CV-subgraph on* Γ $S = (V, E'')$ by choosing E'' as the union of cluster edges and free edges as above, but instead of CC-bridge edges we choose the following edge set:

- **CV-bridge edges:** for each cluster C and each clustered vertex $x \in C' \neq C$, one arbitrary edge $(x, y) \in E$ (if one exists) such that (x, y) is not a cluster edge and $y \in C$. We say that edge (x, y) *connects* x to cluster C.

Theorem 1. *Given a graph* $G = (V, E)$ *and a clustering* Γ*, any CC-subgraph* $S = (V, E')$ *on* Γ *is a (3,2)-spanner of* G*, and any CV-subgraph* $S = (V, E'')$ *on* Γ *is a (2,1)-spanner of* G*.*

3 The Algorithm

We first describe the algorithm for constructing a (3,2)-spanner: the construction of a (2,1)-spanner follows the same lines. In order to build a (3,2)-spanner we maintain the following information. For each vertex x we store:

- a flag indicating whether x is clustered or free. In case it is clustered we also store the label $Cl(x)$ and the label of $center(x)$;
- a set of edges $F(x)$, that contains the set of free edges (x, y) leading to each free vertex $y \in N(x)$. In set $F(x)$ we may possibly have also some edge (x, z) such that z is no longer free. We maintain the value $|F(x)|$;
- if x is the center of a cluster C, the list of vertices belonging to C.

Bridge edges are stored in a matrix *Bridge*, whose entry $Bridge[i, j]$ contains the bridge connecting clusters C_i and C_j. Obviously, $Bridge[i, j]$ is empty in the case $(C_i \times C_j) \cap E = \emptyset$, but it is also possible that $Bridge[i, j]$ is empty if there is some edge in a set $F(x)$ that connects C_i and C_j. For sake of efficiency, we need to bound the number of clusters. Thus, we create a cluster only if it contains at least $n^{1/3}$ vertices, where n is the number of vertices currently known to the algorithm. This implies that clusters created in the first steps of the edge stream processing can be very small, while clusters created later must be larger.

An edge (x, y) is processed as shown in Procedure `CC-EdgeProcess` (see Figure 1). We first check whether edge (x, y) connects two clusters that are

not yet connected by a CC-bridge, and update matrix *Bridge* accordingly. In case (x, y) joins a center x and a free vertex y, we add y to $C(x)$. Otherwise, if y is free, we add (x, y) to the set $F(x)$ of edges that connect x to (possibly) free vertices. When this set becomes too large, we scan $F(x)$ checking whether the other endpoint of each edge is still free: if we find at least $n^{1/3}$ free vertices a new cluster centered at x is created. When vertices are included in a new cluster, as in Lines 3 (resp. 4) of Procedure `CC-EdgeProcess`, we do not update set $F(y)$ for all $y \in N(z)$ (resp. for all $y \in N(x)$). This means that sets $F(\cdot)$ may contain edges whose endpoints are no longer free. For this reason, the condition in Line 1 of Procedure `CC-EdgeProcess` does not ensure that a new cluster centered on x can be created. So, we clean up set $F(x)$ and count the effective number of free vertices in $N(x)$ any time we try to create a new cluster.

It is possible to show that the (3,2)-spanner can be updated in constant amortized time per edge scanned.

Theorem 2. *Each edge is processed by Procedure* `CC-EdgeProcess` *in* $O(1)$ *amortized time.*

```
Procedure CC-EdgeProcess
input: edge (x, y)
    if both x and y are clustered then
        if Cl(x) ≠ Cl(y) and there is no bridge between Cl(x) and Cl(y) then
            store (x, y) into Bridge as the bridge connecting Cl(x) and Cl(y)
        else edge (x, y) is discarded
    if x is a center and y is free (or analogously y is a center and x is free) then
        add y to C(x) and mark y as clustered
        add (x, y) to the set of cluster edges
    if y is free then
        add (x, y) to F(x)
1.      if |F(x)| ≥ 2 · n^{1/3} then
2.          foreach edge (x, z) ∈ F(x)    {clean up set F(x)}
                if z is not free then
                    remove (x, z) from F(x)
                    if x clustered and ¬∃ bridge between Cl(x) and Cl(z) then
                        store (x, z) as the bridge connecting Cl(x) and Cl(z)
            if |F(x)| ≥ n^{1/3} then  {a new cluster centered on x can be created }
3.              create a new cluster C, centered on x, containing
                        all z s.t. (x, z) ∈ F(x) and mark all those z as clustered
                if x is free then
4.                  add x to C and mark x as clustered
                else store one of the cluster edges (x, z) into Bridge
                                as the bridge connecting Cl(x) and C
    if x is free then same as above, with x and y swapped
```

Fig. 1. Procedure `CC-EdgeProcess`

As far as the worst-case time per operation is concerned, we can show the following result:

Theorem 3. *Each edge is processed by Procedure* `CC-EdgeProcess` *in* $O(n^{1/3})$ *worst case time.*

The algorithm proposed computes a (3,2)-spanner of optimal size, as shown by the following theorem.

Theorem 4. *The (3,2)-spanner computed by Procedure* `CC-EdgeProcess` *has* $O(n^{4/3})$ *edges.*

The above theorem relies on the fact the number of clusters is always $O(n^{2/3})$. The algorithm needs to maintain in memory all current spanner edges, plus the matrix *Bridge*, that stores bridges between all pairs of clusters. Thus we can state the following:

Theorem 5. *Algorithm* `CC-EdgeProcess` *needs overall* $O(n^{4/3})$ *space.*

The construction of a (2,1)-spanner follows the same lines of the algorithm above, but we maintain bridges between each cluster and each clustered vertex, namely CV-bridge edges, instead of bridges between pairs of clusters, i.e., CC-bridge edges. Moreover, clusters are created containing at least $n^{1/2}$ (instead of $n^{1/3}$) vertices. The following theorem holds:

Theorem 6. *Given a graph* G*, it is possible to compute a (2,1)-spanner of* G *with* $O(n^{3/2})$ *edges by doing a single pass on the edges in total space* $O(n^{3/2})$*. Each edge is processed in* $O(1)$ *amortized time. The worst-case time required per processed edge is* $O(n^{1/2})$*.*

The algorithms described for $(k, k-1)$-spanners, $k = 2, 3$, need a total of $O(n^{1+1/k})$ space. In the case of very large graphs, this value could exceed the size of the internal memory. If the size of the internal memory is $\Theta(n)$ (semi-external memory model), it is possible to modify the algorithms, so that they maintain only $O(n)$ information in internal memory, while all data structures requiring larger space are maintained in external memory. We refer to the well known external memory model of [24], in which data is organized on external memory in blocks, each of size B, and the algorithm performance is evaluated in terms of the number of I/O operations, i.e., block transfers between external memory and main memory. The only data structures that may require space $\Omega(n)$ are sets $F(\cdot)$ and the matrix *Bridge*. We store each set $F(\cdot)$ in $F|(\cdot)|/B$ blocks, that are scanned and updated in Lines 2 and 3. Each scan requires $O(n^{1/k}/B)$ I/Os, and the total number of I/Os during the whole sequence of edge processing is $O(m/B)$ (in the realistic assumption that $n^{1/k} > B$). In order to efficiently update and query the matrix *Bridge* we proceed as follows. Any time we are going to discard an edge (x, y), we must check whether $Cl(x)$ and $Cl(y)$ are already connected by a CC-bridge. We do not need to answer the query in real-time, since this test does not affect the behaviour of the algorithm. So, we append (x, y) to a set $Bridge_{temp}$ of potential CC-bridges to be processed; items in $Bridge_{temp}$ are written in any order in blocks on external memory. When $Bridge_{temp}$ contains

more than $n^{4/3}$ edges, we lexicographically sort $Bridge_{temp}$ according to the pair of cluster labels of the endpoints (by two passes of radix sort) and check each candidate bridge against the sorted set of items in matrix $Bridge$, obtaining the updated (sorted) version of $Bridge$. All this process requires $O(n^{4/3}/B)$ I/Os, and is performed each $\Omega(n^{4/3})$ edges. We can thus state the following results:

Theorem 7. *Given an unweighted graph G, a $(k, k-1)$-spanner of G with $O(n^{1+1/k})$ edges, for $k = 2, 3$, can be computed in the data streaming model by a single pass using $O(n)$ internal memory and $O(n^{1+1/k}/B)$ external memory blocks, assuming $n^{1/k} > B$, requiring:*

- *$O(1)$ amortized time in internal memory plus $O(1/B)$ amortized I/O's to process each edge;*
- *$O(n^{1/k})$ worst-case time in internal memory plus $O(n^{1/k}/B)$ worst case I/O's to process each edge.*

4 Experiments

In this section, we compare experimentally the algorithm for constructing a (3,2)-spanner described in Section 3 with the randomized streaming algorithm for $(2k-1)$-spanners by Baswana [7], for $k \geq 2$: this algorithm was a natural choice since it has the best theoretical performances known so far and it is likely to outperform previous results also in practice due to its simpler data structures. Our goal is to study the differences between the two techniques in terms of both quality of solutions and time/space requirements. To assess the performance of the two algorithms, we have also considered an off-line method for constructing a $(2k-1)$-spanner of an unweighted graph suggested by Zwick [25]. Since this algorithm is able to access edges in any order, we expect that it could build spanners of better quality (with respect to size and/or average stretch) than those computed by any streaming algorithm, and so we use it as a benchmark in our experiments.

Experimental Setup. We ran our experiments on a PC equipped with a 3 GHz Intel Pentium 4 dual core processor, 2 MB L2 cache, 2 GB RAM, using a 50 GB Ext3 partition on a 250 GB IDE hard disk running Linux Mandriva 2007 kernel 2.6.17. All algorithms were coded in C using a uniform programming style and common data structures. Programs were compiled with `gcc 4.1.1` with optimization flag `-O4`. The performance measures we considered are:

- *Estimates of average stretch, stretch variance, and maximum stretch*: stretches are measured as the ratio between distances in the spanner and in the original graph on a sample of vertex pairs: from 100 random sources to all other vertices. These estimates proved to be very close to the exact values obtained by running an all-pairs shortest paths implementation, but much faster to compute.
- *Spanner size*: number of edges in the spanner computed by the algorithm.

- *Running time*: total time in seconds required to compute a spanner (over the whole stream).
- *Memory footprint*: peak amount of memory in megabytes required by the algorithm during its execution.

Each measured value reported in our charts was obtained as the average of at least three independent trials on the same data point. In our experiments, we considered the following synthetic graph families:

- *Power Law graphs*: small world graphs based on the recursive matrix (R-MAT) model [12].
- *Random graphs*: based on the Erdős-Rényi $G_{n,p}$ model [16].
- *Clustered graphs*: graphs made of clusters connected to form a line. Each cluster is a random $G_{n,p}$ graph. These graphs tend to have higher diameter than the power law and random graphs.

Graphs were presented to the algorithms as an on-line stream of edges in random order. Since we observed similar trends of the algorithm performances on all the test sets we considered, in this extended abstract we only report the results for power law graphs, which are especially relevant in applications. Our experimental package, including the instance generators, the complete C source codes of our implementations, and additional charts summarizing the results for other classes of input graphs not reported in this paper can be found at [1].

The Algorithm by Ausiello, Franciosa, and Italiano (`AFI`). In our implementation, we modified the original algorithm of Figure 1 (which we denote by `AFI`) by parameterizing it with a user-defined *clustering threshold* c that replaces the adaptive $n^{1/3}$ value used by the algorithm as a threshold to form clusters, where n is the current number of vertices encountered in the stream. As we will see, this allows us to produce spanners of different size and stretch, leading to a more flexible algorithm in practice. The charts of Figure 2 show how spanner size, average stretch, running time, and memory footprint depend upon the clustering threshold on a power law graph with $n = 8,192$ and $m = 13,003,600$. Observe that the number of edges in the spanner initially drops down with the clustering threshold c, reaches a minimum around $c = 50$, and then grows up again for higher values of c. Intuitively, this can be explained by the fact that for small values of c the algorithm forms many clusters, and therefore there will be many CC-bridge edges. On the other hand, if the threshold is high there will be many free vertices, and sets $F(\cdot)$ can be larger. The running time and the memory footprint follow the same trend, while the smaller the spanner, the higher the stretch factor. Notice that the value of the clustering threshold that minimizes the spanner size minimizes also the running time and the space usage. Interestingly, while this optimal value may depend upon n and m, the adaptive choice of $c = n^{1/3}$ made in the original algorithm, plotted as dashed horizontal lines in the charts of Figure 2, seems to be very close to it. We also notice that the spanner size can be much smaller in practice than the theoretical bound

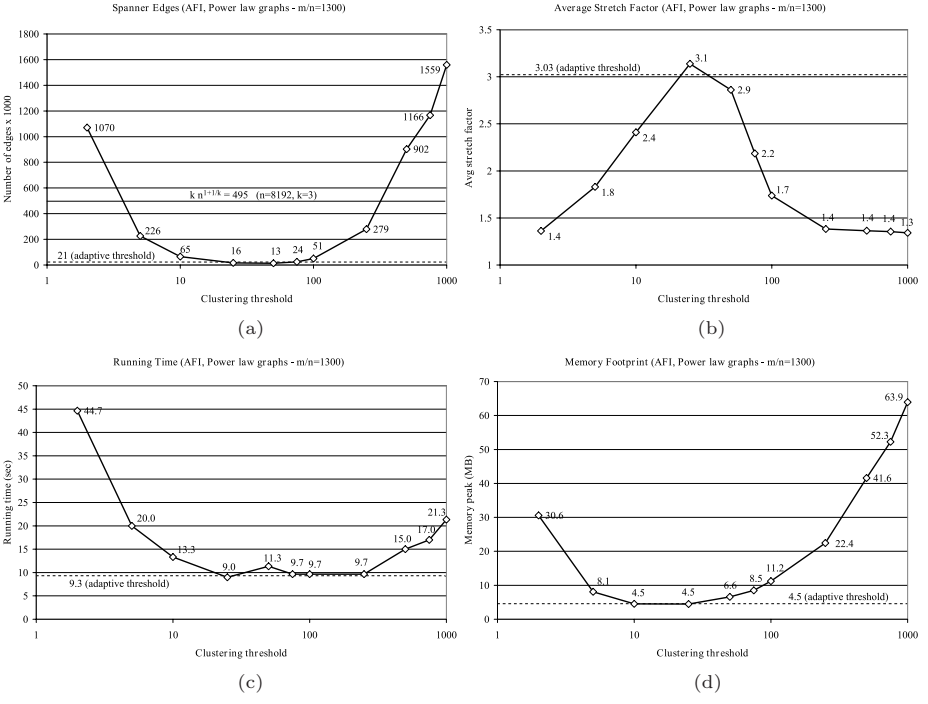


Fig. 2. Analysis of algorithm `AFI` on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of the clustering threshold: (a) number of edges in the spanner; (b) average stretch estimate; (c) running time in seconds; (d) memory peak in megabytes

$k \cdot n^{1+1/k}$ as shown in Figure 2 (a), suggesting that multiplicative factors in `AFI`'s $O(n^{1+1/k})$ bound on the number of spanner edges can be small.

The Algorithm by Baswana (B). We analyze experimentally a very recent randomized streaming algorithm by Baswana [7], which computes a $(2k-1)$-spanner of expected size $O(k \cdot n^{1+1/k})$ in one pass in $O(m+n)$ total time, using $O(k \cdot n^{1+1/k})$ memory space. We denote this algorithm by `B`.

In Figure 3 we show how spanner size, average stretch, running time, and memory footprint depend upon parameter k on a power law graph with $n = 8,192$ and $m = 13,003,600$. We first notice that the number of edges in the spanner follows the theoretical trend of $k \cdot n^{1+1/k}$ for small values of k, but it quickly gets flattened out for $k > 10$. Similarly to what we have seen for `AFI`, the smaller the spanner, the higher the average stretch. However, differently from `AFI`, the running time is minimized for the value of k that maximizes the spanner size, i.e., $k = 2$. Notice that, while increasing k beyond 100 does not yield any spanner size or stretch benefits, it raises the memory consumption considerably. On all test sets we considered, the algorithm seems to be of practical value only for small values of k.

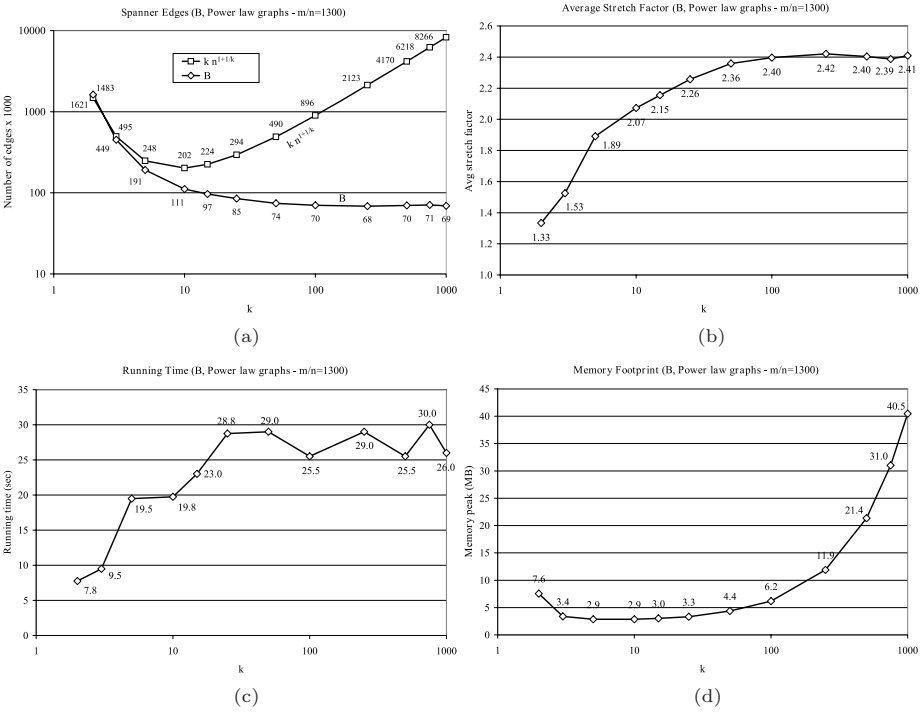


Fig. 3. Analysis of algorithm B on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of k: (a) number of edges in the spanner; (b) average stretch estimate; (c) running time in seconds; (d) memory peak in megabytes

The Algorithm by Zwick (Z). We now describe a simple off-line algorithm due to Zwick [25], which we denote by Z. Given an unweighted undirected graph with n vertices and m edges, it computes in $O(m+n)$ time a $(2k-1)$-spanner with $n^{1+1/k}$ edges for any integer $k \geq 2$. We use this algorithm as a benchmark in the performance analysis of AFI and B. Let $N_i(v)$ be the vertices that are at distance exactly i from v. The algorithm works by alternating two phases. In the first phase, it chooses an arbitrary vertex v, starts from $N_0(v) = \{v\}$, and computes $N_i(v)$ for $i = 1, 2, 3, \ldots$, by finding all unreached vertices that are one hop away from a vertex in $N_{i-1}(v)$, until $|N_i(v)| \leq n^{1/k} \cdot |N_{i-1}(v)|$. This must happen for some $i \leq k$, as otherwise $|N_k(v)| > n$, which is impossible. In the second phase, the algorithm builds a shortest path tree from v to all vertices at distance at most i from v, and adds the edges of this tree to the spanner. Then it removes from the graph the vertices at distance at most $(i-1)$ from v, and all edges touching them. Note that when an edge is removed there is a path of length at most $(2k-1)$ in the spanner passing through v. These two phases are repeated until no vertices remain. It is easy to see that the total running time is linear, and that the subgraph constructed is a $(2k-1)$-spanner of the original graph having $O(n^{1+1/k})$ edges.

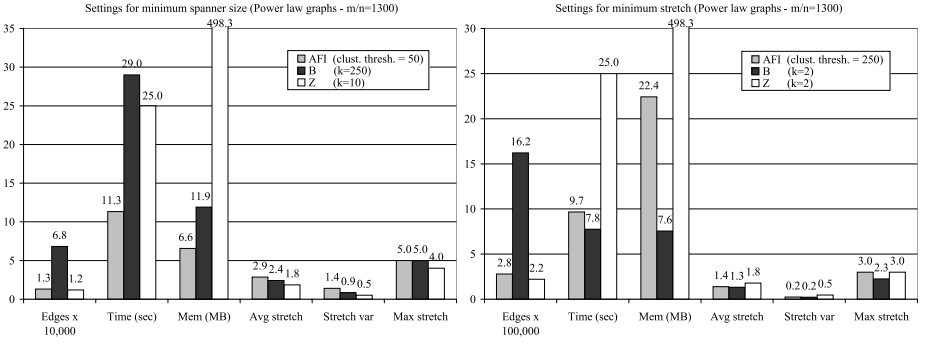


Fig. 4. Comparison of algorithms `AFI`, `B`, and `Z` on a power law graph with $n = 8,192$ and $m = 13,003,600$ for the values of k (for `B` and `Z`) and of the clustering threshold (for `AFI`) that yield the smallest spanner (left) and the smallest stretch (right)

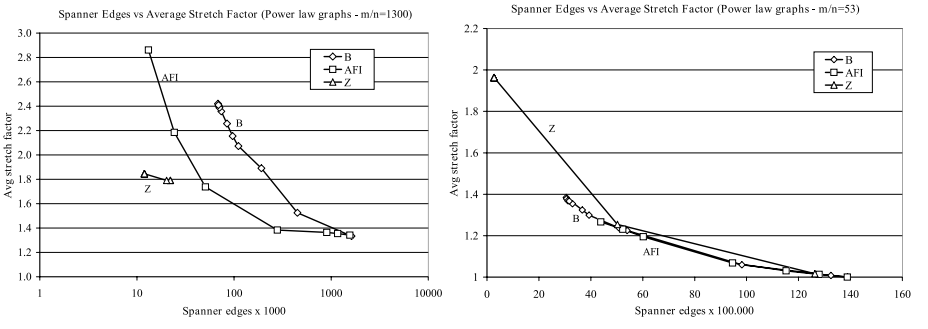


Fig. 5. Comparison of the solution quality achieved by algorithms `AFI`, `B`, and `Z` on a power law graph with $n = 8,192$ and $m = 13,003,600$ (left) and on a power law graph with $n = 262,144$ and $m = 13,927,365$ (right) for different values of k (for `B` and `Z`) and of the clustering threshold (for `AFI`)

Discussion. Figure 4 shows an overall comparison of algorithms `AFI`, `B`, and `Z` on a power law graph with $n = 8,192$ and $m = 13,003,600$ for two choices of parameters k (for `B`, and `Z`) and c (for `AFI`): the setting that minimizes the spanner size (chart on the left hand side) and the setting that minimizes the stretch factor (chart on the right hand side). We first observe that in both cases, `B` found much larger spanners than `AFI`, yielding however slightly better average stretches. To minimize the spanner size, `AFI` was faster and used less memory than `B`. Conversely, to minimize the stretch, `B` was faster and used less memory than `AFI`. Both algorithms used substantially less memory than `Z`, which needs to keep the whole graph in internal memory. They were also able to find spanners with smaller stretch than `Z`, and this is somewhat surprising given that `Z` is an off-line algorithm.

Figure 5 focuses on the quality of the spanners found by `AFI`, `B`, and `Z` on two power law graphs with different edge densities. The charts plot size and stretch

of the spanners that can be achieved by considering the full range of parameters k and c. Ideally, we would like to find a spanner with the smallest stretch and the smallest size simultaneously. In the case of the graph with $n = 8,192$ and $m = 13,003,600$, `Z` yielded the best spanners, and `AFI` found spanners of better quality than `B` (chart on the left hand side). For the sparser power law graph with $n = 262,144$ and $m = 13,927,365$ (chart on the right hand side) the differences between the algorithms were instead negligible.

In summary, our experiments seem to suggest that:

- Algorithms `AFI` and `B` are likely to be very efficient in practice, as they find spanners of size and stretch much smaller than the theoretical bounds, and are competitive with off-line algorithms.
- Setting large values of k in `B` does not produce sparser spanners in all test sets we have considered, so small values of k, for which `AFI` was designed, seem to be the case of interest in practice.
- `AFI`, in addition to being deterministic and not requiring prior knowledge of the number of vertices in the graph, seems to have the additional benefit of producing spanners of better quality than `B`, while still using a comparable amount of time and space resources.

References

1. Experimental package. See: http://www.dis.uniroma1.it/~ribichini/spanner/
2. Aggarwal, G., Datar, M., Rajagopalan, S., Ruhl, M.: On the streaming model augmented with sorting primitive. In: FOCS'04, pp. 540–549 (2004)
3. Althofer, I., Das, G., Dobkin, D.P., Joseph, D., Soares, J.: On sparse spanners of weighted graphs. Discrete & Computational Geometry 9, 81–100 (1993)
4. Ausiello, G., Franciosa, P.G., Italiano, G.F.: Small stretch spanners on dynamic graphs. Journal of Graph Algorithms and Applications 10(2), 365–385 (2006)
5. Awerbuch, B.: Complexity of network synchroniz. JACM 32(4), 804–823 (1985)
6. Baswana, S.: Dynamic algorithms for graph spanners. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 76–87. Springer, Heidelberg (2006)
7. Baswana, S.: Faster streaming algorithms for graph spanners: (2006), http://www.citebase.org/abstract?id=oaiarXiv.org:cs/0611023
8. Baswana, S., Kavitha, T., Mehlhorn, K., Pettie, S.: New constructions of (α, β)-spanners and purely additive spanners. In: SODA'05, pp. 672–681 (2005)
9. Baswana, S., Sen, S.: A simple linear time algorithm for computing $(2k-1)$-spanner of $O(n^{1+1/k})$ size for weighted graphs. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 384–396. Springer, Heidelberg (2003)
10. Cai, L.: NP-completeness of minimum spanner problems. Discr. Appl. Math. and Combinat. Operations Res. and Comp. Science 48(2), 187–194 (1994)
11. Cai, L., Keil, J.M.: Degree-bounded spanners. Par. Proc. Lett. 3, 457–468 (1993)
12. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: 4th SIAM International Conference on Data Mining (2004)
13. Chew, L.P.: There are planar graphs almost as good as the complete graph. Journal of Computer and System Sciences 39(2), 205–219 (1989)

14. Das, G., Joseph, D.: Which triangulations approximate the complete graph? In: Djidjev, H.N. (ed.) Optimal Algorithms. LNCS, vol. 401, pp. 168–192. Springer, Heidelberg (1989)
15. Dobkin, D., Friedman, S.J., Supowit, K.J.: Delaunay graphs are almost as good as complete graphs. Discrete & Computational Geometry 5, 399–407 (1990)
16. Erdős, P., Rényi, A.: On random graphs. P. Math. Debrecen 6, 290–291 (1959)
17. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: Graph distances in the streaming model: the value of space. In: SODA'05, pp. 745–754 (2005)
18. Liestman, A.L., Shermer, T.: Grid spanners. Networks 23, 122–133 (1993)
19. Liestman, A.L., Shermer, T.: Additive graph spanners. Networks, 23, 343–364 (1993)
20. Peleg, D., Shäffer, A.: Graph spanners. J. Graph Theory 13, 99–116 (1989)
21. Peleg, D., Ullman, J.D.: An optimal synchronizer for the hypercube. SIAM Journal on Computing 18(4), 740–747 (1989)
22. Richards, D., Liestman, A.L.: Degree-constrained pyramid spanners. JPDC: Journal of Parallel and Distributed Computing 25, 1–6 (1995)
23. Roditty, L., Thorup, M., Zwick, U.: Deterministic constructions of approximate distance oracles and spanners. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 261–272. Springer, Heidelberg (2005)
24. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Computing Surveys 33(2), 209–271 (2001)
25. Zwick, U.: Personal communication

Estimating Clustering Indexes in Data Streams*

Luciana S. Buriol[1], Gereon Frahling[2], Stefano Leonardi[3], and Christian Sohler[4]

[1] Federal University of Rio Grande do Sul, Porto Alegre, Brazil
buriol@inf.ufrgs.br
[2] Google Research, New York, United States
gereon@google.com
[3] University of Rome "La Sapienza", Rome, Italy
Stefano.Leonardi@dis.uniroma1.it
[4] Heinz Nixdorf Institute and University of Paderborn, Paderborn, Germany
csohler@upb.de

Abstract. We present random sampling algorithms that with probability at least $1 - \delta$ compute a $(1 \pm \epsilon)$-approximation of the clustering coefficient and of the number of bipartite clique subgraphs of a graph given as an incidence stream of edges. The space used by our algorithm to estimate the clustering coefficient is inversely related to the clustering coefficient of the network itself. The space used by our algorithm to compute the number $K_{3,3}$ of bipartite cliques is proportional to the ratio between the number of $K_{1,3}$ and $K_{3,3}$ in the graph.

Since the space complexity depends only on the structure of the input graph and not on the number of nodes, our algorithms scale very well with increasing graph size. Therefore they provide a basic tool to analyze the structure of dense clusters in large graphs and have many applications in the discovery of web communities, the analysis of the structure of large social networks and the probing of frequent patterns in large graphs.

We implemented both algorithms and evaluated their performance on networks from different application domains and of different size; The largest instance is a webgraph consisting of more than 135 million nodes and 1 billion edges. Both algorithms compute accurate results in reasonable time on the tested instances.

1 Introduction

The analysis of the structure of large networks often requires the computation of network indexes based on the number of certain small subgraphs. Much attention has recently been devoted to the structural analysis of networks arising in information systems; Physical telecommunication connections, overlay networks, and software systems are some examples. The observation of certain dense subgraphs in the webgraph, the graph formed by web pages and hyperlinked connections, has also been considered in the attempt of tracing the emergence of hidden cyber-communities [12]. The counted subgraphs are typically dense bipartite cliques of small size that are interpreted as cores

* This work was partially supported by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS) and DFG project So 514/1-1.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 618–632, 2007.

of web communities: The vertices in the left partition of the bipartite clique are considered as member pages pointing to a set of centers/authorities for the community. A large number of these subgraphs has been observed in large crawls of the Web by Kumar et al. [12] and by Laura et al. [14]. A stochastic model of graphs that resembles the structure of the Web [13], called the *copying model*, uses these dense subgraphs as building blocks of the process of network formation of the Webgraph. Counting the number of such subgraphs therefore provides deeper insight into the construction process of large graphs and can provide justifications for different construction models.

Another network index widely used in the analysis of complex network structures is the *clustering coefficient* [18]. It is defined as the normalized sum of the fraction of neighbor pairs of a vertex that are connected. It measures the degree at which clusters decompose into communities [6]. See Section 3 for an exact definition of the clustering coefficient.

Estimating the value of network indexes in large graphs is a challenging computational task. Current state of the art methods are either computationally unfeasible on large data sets or do not provide guarantees on the accuracy of the estimation. The best known methods for the solution of the simplest non trivial version of this problem, i.e. counting the number of triangles in a graph, use matrix multiplication [4]. Even on graphs of medium size this is not computationally feasible because of both the time complexity and the main memory usage required to store the whole graph. Schank and Wagner [17] gave an extensive experimental study of algorithms counting and listing triangles and computing the clustering coefficient in small and medium size graphs. Their algorithms work in main memory and report results for graphs up to 668 thousand nodes.

A natural way to address massive data sets consisting of more than 100 million nodes is the data stream model [10,15]. In this model data arrives in a stream, one item at a time, and the algorithms are required to use very little space and per-item processing time. Data stream algorithms are able to sequentially read the data from secondary memory storage devices and avoid slow random access to the data. Even massive data sets which do not fit on any available storage device could be handled.

Data stream algorithms have been proposed for various problems. Examples are the computation of frequency moments [1], histograms [9], or Wavelet transforms [8]. The large body of work on data stream algorithms contrasts with a lack of efficient solutions of natural graph problems in the streaming model of computation [10]. Bar-Yossef, Kumar and Sivakumar [19] gave a first solution for counting triangles in the data stream model. They considered both the "adjacency stream model" where the graph is presented as a sequence of edges in arbitrary order and the "incidence stream" model where they consider only bounded-degree graphs and all edges incident to a vertex are presented successively. Their algorithms provide an ϵ approximation with probability $1 - \delta$ using a number of memory cells in some cases smaller than a naive sampling technique algorithm. The algorithms are obtained through a so called "list" efficient reduction to the problem of computing frequency moments [1]. Subsequently, more algorithms to count triangles have been developed for the adjacency stream model [11,2], and the incidence stream model [2].

Schank and Wagner [17] gave an algorithm, which returns with probability $1-\delta$ a $(1\pm\epsilon)$-approximation on the clustering coefficient C_G of a graph G when the graph is given as an incidence stream. It needs two passes over the data and $\mathcal{O}(\log\frac{1}{\delta}\cdot\frac{1}{\epsilon^2\cdot C_G})$ memory cells. We will recap the algorithm in Section 4.

2 Our Results

In this paper we present random sampling data stream algorithms to compute the clustering coefficient and the number of bipartite cliques in graphs given as an incidence stream.

We improve the 2-pass streaming algorithm of [17] to estimate the clustering coefficient C_G. Our new algorithm only requires one pass over the stream of edges. Although the memory needed by our improved algorithm is slightly larger ($\mathcal{O}(\frac{\log(1/\delta)\cdot\log(|V|)}{\epsilon^2 C_G}\cdot\log\frac{1}{\epsilon\delta C_G}$) memory cells compared to $\mathcal{O}(\log\frac{1}{\delta}\cdot\frac{1}{\epsilon^2\cdot C_G})$ memory cells needed by the 2-pass algorithm), our technique enables the application of the algorithm in real streaming scenarios. Distributed crawlers collecting Web pages and their links could perform structural analysis of the Webgraph online while transfering the data to storage devices. Our new algorithm is also applicable to graphs which are too large to be stored at all.

We also provide a data stream algorithm that computes a $(1\pm\epsilon)$- approximation of the number of $K_{3,3}$ (bipartite cliques having three nodes in each partition) in a graph given as an incidence stream ordered by destination nodes with outdegree bounded by Δ. The algorithm needs $O\left(\log(|V|)\cdot\frac{K_{3,1}\cdot\Delta^2\ln(\frac{1}{\delta})}{K_{3,3}\cdot\epsilon^2}\right)$ memory cells. Previous techniques to count subgraphs [11,12] could only be applied to the estimation of subgraphs containing a star (triangles or complete cliques K_i for example). Our algorithm is based on a new sampling technique, which can be extended to estimate the number of bipartite cliques $K_{i,j}$ for arbitrary i and j. Since the space complexities of our algorithms depend only on the structure of the input graph (and not on it's size), they can be used to estimate these structural properties even for large webgraph crawls.

We present optimized implementations of the algorithms and tests on networks including large web-graphs, graphs of the largest online encyclopedia Wikipidia [3], and graphs of collaborations between actors and authors. Our results show that for all networks a relatively small number of memory cells already suffices to provide good approximations of the clustering coefficient and the number of bipartite cliques.

2.1 Structure of the Paper

We present in Section 4 the algorithm to estimate the clustering coefficient and in Section 5 the algorithm to count $K_{3,3}$ cliques. Section 6 introduces the optimized implementations of the algorithms which are tested on real data sets in Section 7.

3 Preliminaries

Let $G=(V,E)$ denote a directed graph without self-loops. We assume that G is given as a stream of incidence lists, one incidence list $\mathcal{L}(v)$ for each node $v\in V$.

The incidence list $\mathcal{L}(v)$ of a node v consists of all edges that are directed to v, i.e. all edges $e \in E$ of the form $e = [u, v\rangle$ for some $u \in V$. The incidence lists can appear in arbitrary order in the stream. We also do not assume a particular order of the edges of an incidence list. When we consider undirected graphs, we simply assume that every edge is represented by two undirected edges, appearing in the incidence lists of both nodes.

There are natural motivations for the model of incidence lists. Big graphs as the webgraph are often retrieved by crawlers who essentially produce streams of incidence lists. Furthermore even a stream of all known edges in the webgraph can be transformed into a stream of incidence lists by one round of a parallel computing environment such as Googles MapReduce (the complete process is described in [5]).

Definition of Clustering Coefficient. Let $G = (V, E)$ be an undirected graph. For every vertex $v \in V$ let $\mathcal{N}(v)$ denote its neighborhood, i.e. $\mathcal{N}(v) = \{u \in V : \exists (u, v) \in E\}$. The *clustering coefficient* C_v of a vertex $v \in V$ of G is defined as the probability that a random pair of its neighbors is connected by an edge, i.e.

$$C_v := \frac{\left|\left\{(u, w) \in E : u \in \mathcal{N}(v) \text{ and } w \in \mathcal{N}(v)\right\}\right|}{\binom{|\mathcal{N}(v)|}{2}}.$$

In case of $|\mathcal{N}(v)| < 2$ we define $C_v := 0$. The *clustering coefficient* C_G of G is the average clustering coefficient of its vertices.

4 Approximating the Clustering Coefficient

In this section we recapitulate how to approximate the Clustering Coefficient using an algorithm from [17], which we state only for the unweighted case.

APPROXCLUSTERINGCOEFFICIENT(G, s)
  sample s vertices $w_1, \ldots, w_s$ uniformly at random
  for $i = 1$ **to** s **do**
    sample a random pair (u, v), $u \neq v$, of points uniformly from $\mathcal{N}(w_i)$
    if $(u, v) \in E$ **then set** $X_i \leftarrow 1$
    else set $X_i \leftarrow 0$
  Output $X := \frac{1}{s} \cdot \sum_{i=1}^{s} X_i$

It is easy to see that the algorithm can be implemented in two passes over the data. One pass to select the random vertices and the random pairs of neighbors and another pass to check for each pair of neighbors whether they are connected by an edge. For sake of completeness we give an analysis of the algorithm below. We first show that the expected value of X_i is exactly C_G. We have for each $i \in \{1, \ldots, s\}$:

$$\mathbf{E}[X_i] = \frac{1}{n} \cdot \sum_{v \in V} \mathbf{E}[X_i \mid w_i = v] = \frac{1}{n} \cdot \sum_{v \in V} C_v = C_G.$$

Then we use the fact that for $0-1$ random variables we have

$$\mathbf{Var}[X_i] \leq \mathbf{E}[X_i^2] = \mathbf{E}[X_i] = C_G.$$

Now we analyze the variance of X. Since the X_i are mutually independent we get

$$\mathbf{Var}[X] = \mathbf{Var}[\frac{1}{s} \cdot \sum_{i=1}^{s} X_i] = \frac{1}{s^2} \cdot \sum_{i=1}^{s} \mathbf{Var}[X_i] \leq \frac{C_G}{s}.$$

Finally, we can apply Chebyshev inequality. This gives us

$$\mathbf{Pr}\big[\big|X - \mathbf{E}[X]\big| \geq \epsilon \cdot \mathbf{E}[X]\big] \leq \frac{\mathbf{Var}[X]}{(\epsilon \cdot \mathbf{E}[X])^2} \leq \frac{C_G}{s \cdot \epsilon^2 \cdot C_G^2} = \frac{1}{s \cdot \epsilon^2 \cdot C_G}.$$

If $s \geq \frac{3}{\epsilon^2 \cdot C_G}$ then with probability $2/3$ the algorithm ApproxClusteringCoefficient approximates the clustering coefficient of G within a relative error of $(1 \pm \epsilon)$. A standard success amplification technique (running the algorithm $\Theta(\log \frac{1}{\delta})$ times and returning the median of all results) leads to the following corollary, which follows immediately from [17][Theorem 1].

Corollary 1. *There is a 2-pass streaming algorithm which with probability $1-\delta$ returns a $(1 \pm \epsilon)$-approximation on the clustering coefficient C_G of a graph G when the graph is given as a incidence stream. It needs $\mathcal{O}(\log \frac{1}{\delta} \cdot \frac{1}{\epsilon^2 \cdot C_G})$ memory cells.*

4.1 A One-Pass Algorithm

In this section we show that it is possible to reduce both passes of the algorithm above to one pass over the incidence stream. Again we sample a vertex w uniformly at random, pick two of its neighbors uniformly at random, and check whether these neighbors are connected by an edge.

To pick two random neighbors of w we use random hash functions in a way somewhat similar to random sampling in dynamic data streams [7]. We will require a guess 2^j of the degree of w for each value of $j \in \{0, \ldots, \lceil \log n \rceil\}$. For each guess we pick a random hash function $h_j : V \rightarrow \{1, ..., 2^j\}$. Since we can count the degree of w while passing over the data stream, we can at the end of our algorithm pick the right value of j (having approximately $2^j = \text{degree}(w)$) and forget all other values of j. For the right value of j the hash function will map with constant probability exactly two vertices from the neighborhood $\mathcal{N}(w)$ of w to the value 1, i.e. $|h_j^{-1}(1) \cap \mathcal{N}(w)| = 2$. Conditioned on this event, these two vertices are distributed uniformly at random among $\mathcal{N}(w)$. They will be our two sampled vertices. The algorithm outputs a random variable X having expected value C_G. In case we do not have exactly two neighbors of w mapped to 1 by h, it outputs an error ($\perp$). We assume fully random hash functions.

In the algorithm u_j, v_j are random variables for the first and second neighbor x of w having $h_j(x) = 1$. The variable X_j denotes the output value for $j = \lceil \log d \rceil$, where d is the degree of w.

```
ONEPASSCLUSTERINGCOEFFICIENT
    sample a vertex w uniformly at random
    for j = 1 to ⌈log V⌉ do
        X_j ← ⊥; u_j ← ⊥; v_j ← ⊥
        h_j ← random hash function h : V → {1, ..., 2^j}
    for each incidence list L(x) in the stream do
        for j = 1 to ⌈log V⌉ do
            if h_j(x) = 1 and w ∈ L(x) then                  // (x ∈ N(w) will be sampled)
                if u_j = ⊥ then u_j ← x                      // (x is first sampled neighbor of w)
                else
                    if v_j = ⊥ then
                        v_j = x                              // (x is second sampled neighbor of w)
                        if u_j ∈ L(x) then X_j ← 1           // (check, if there is edge between u_j and v_j)
                        else X_j ← 0
                    else X_j ← ⊥                             // (|h^{-1} ∩ N(w)| > 2)
        if x = w then d ← |L(x)|                             // (set the degree of w to the right value)
        if d < 2 then output 0
        if d ≥ 2 then output X_{⌈log d⌉}
```

Theorem 1. *With probability* $\frac{1}{44}$ *the algorithm* ONEPASSCLUSTERINGCOEFFICIENT *does not output* $\perp$. *If it does not output* $\perp$ *it outputs a* $0-1$ *random variable* X *having expected value* $\mathbf{E}[X] = C_G$.

Proof. Let d be the degree of node w and $\tilde{d} = 2^{\lceil \log d \rceil}$. We have $d \leq \tilde{d} \leq 2 \cdot d$. Let be $h := h_{\lceil \log d \rceil}$, $u := u_{\lceil \log d \rceil}$, and $v := v_{\lceil \log d \rceil}$. At the end the algorithm chooses the result $X_{\lceil \log d \rceil}$. The algorithm outputs $X \neq \perp$ iff exactly 2 nodes adjacent to w are hashed to 1 by h. In that case these two nodes are chosen uniformly at random. The first of these nodes is stored in variable u, the second in variable v. Finally $X_{\lceil \log d \rceil}$ is set to one iff the incidence list of v contains node u, so iff w, u, and v form a triangle.

It remains to show that with probability $1/44$ exactly two neighbouring nodes are hashed to 1 by h. There are d nodes adjacent to w, each one is hashed to 1 by h with probability $\frac{1}{\tilde{d}}$. Let $x_1, ..., x_d$ be the neighbouring nodes of w. If $d < 2$ the algorithm always outputs the correct value $X = 0$. For $d \geq 2$ we obtain:

$$
\begin{aligned}
&\mathbf{Pr}\big[|\{x \in \mathcal{N}(w) | h(x) = 1\}| = 2\big] \\
&= \sum_{i=1}^{d-1} \frac{1}{\tilde{d}} \cdot \mathbf{Pr}\big[\forall_{x \in \{x_1,...,x_{i-1}\}} h(x) \neq 1 \wedge |\{x \in \{x_{i+1}, ..., x_d\} | h(x) = 1\}| = 1\big] \\
&= \sum_{i=1}^{d-1} \frac{1}{\tilde{d}} \cdot \left(1 - \frac{1}{\tilde{d}}\right)^{i-1} \cdot \sum_{j=i+1}^{d} \frac{1}{\tilde{d}} \cdot \left(1 - \frac{1}{\tilde{d}}\right)^{d-i-1} = \frac{1}{\tilde{d}^2} \left(1 - \frac{1}{\tilde{d}}\right)^{d-2} \cdot \sum_{i=1}^{d-1} (d-i) \\
&= \frac{1}{\tilde{d}^2} \cdot \frac{d(d-1)}{2} \cdot \left(1 - \frac{1}{\tilde{d}}\right)^{d-2} \geq \frac{d(d-1)}{8d^2} \cdot \left(1 - \frac{1}{d}\right)^{d} \geq \frac{1}{8} \left(1 - \frac{1}{d}\right) \cdot \frac{1}{e} \geq \frac{1}{16e} \geq \frac{1}{44}
\end{aligned}
$$

To approximate the clustering coefficient in one pass we start $s \cdot r$ instances of ONEPASSCLUSTERINGCOEFFICIENT for $s = \Theta(\log \frac{1}{\delta} \cdot \frac{1}{\epsilon^2 \cdot C_G})$ and $r = \log \frac{2s}{\delta} / \log \frac{44}{43}$. We devide the instances into s groups of r instances.

Fix one group. We bound the probability that all instances in the group return $\perp$: For each instance the probability to return $\perp$ is bounded by $43/44$, therefore we have a probability of at most $(\frac{43}{44})^r = \frac{\delta}{2s}$ for the event that all instances of the group return $\perp$. If this is the case for one of our groups, our algorithm fails. Since we have s groups, each of them failing with probability at most $\frac{\delta}{2s}$, our algorithm fails with probability at most $\frac{\delta}{2}$ by the union bound.

We now consider the case that all groups have at least one instance reporting a value 0 or 1. This case happens with probability at least $1 - \frac{\delta}{2}$. We take the result of the first non-failing instance of each group. Then we have s independent $\{0, 1\}$-variables, each one having expected value C_G. Therefore we can proceed exactly as in the two-pass case ($\Theta(\log \frac{1}{\delta})$ times averaging $\frac{3}{\epsilon^2 \cdot C_G}$ results and taking the median of these averaged results). As shown in the two-pass case this leads to a $(1 \pm \epsilon)$-approximation with probability $1 - \delta$.

Theorem 2. *There is a 1-pass streaming algorithm which returns with probability* $1-\delta$ *a* $(1 \pm \epsilon)$*-approxi-mation of the clustering coefficient* C_G *of a graph* G *when the graph is given as an incidence stream. It uses* $\mathcal{O}(\frac{\log(1/\delta) \cdot \log(|V|)}{\epsilon^2 C_G} \cdot \log \frac{1}{\epsilon \delta C_G})$ *memory cells.* □

5 Counting $K_{3,3}$

In this section we consider the problem to estimate the number of complete bipartite subgraphs $K_{3,3}$ in a directed graph G of bounded outdegree Δ. We assume that the graph is given as a stream of incidence lists ordered by destination nodes (as described in the Preliminaries). Let $K_{3,3}$ denote the number of $K_{3,3}$ subgraphs and $K_{3,1}$ denote the number of $K_{3,1}$ subgraphs. We first assume that we have a method to choose a $K_{3,1}$ uniformly at random.

$K_{3,3}$COUNTING
Choose a $K_{3,1}$ uniformly at random from all $K_{3,1}$ in G.
Let the three edges of the $K_{3,1}$ be $[a, u\rangle$, $[b, u\rangle$ and $[c, u\rangle$
Pass over the stream and stop when all 3 edges of the $K_{3,1}$ are read
Select x_1, x_2 uniformly at random from $\{a, b, c\}$
Select k_1, k_2 uniformly at random from $\{1, 2, \ldots \Delta\}$
if $k_1 = k_2 \wedge x_1 = x_2$ **then** output $\beta = 0$
Continue to pass over the stream.
Select the k_1-th outgoing edge of x_1 (counting from the stop). Call this edge $[x_1, v\rangle$.
Select the k_2-th outgoing edge of x_2 (counting from the stop). Call this edge $[x_2, w\rangle$.
Once $[x_1, v\rangle$ has been selected: check, if $[a, v\rangle$,$[b, v\rangle$,$[c, v\rangle$ are present in the remaining stream
Once $[x_2, w\rangle$ has been selected: check, if $[a, w\rangle$,$[b, w\rangle$,$[c, w\rangle$ are present in the remaining stream
if both is the case **then** output $\beta = 1$ **else** output $\beta = 0$.

Lemma 1. *Algorithm* $K_{3,3}$COUNTING *outputs a random value* β *having expected value*

$$\mathbf{E}[\beta] = \frac{2 \cdot K_{3,3}}{9 \cdot \Delta^2 \cdot K_{3,1}}$$

Proof. Let $K = (A, B, C, U, V, W)$ be an arbitrary fixed $K_{3,3}$ with edges directed from A,B,C to U,V and W. Let X_K denote the indicator random variable for the event that $\beta = 1$ with witness K. We would like to determine the probability for $X_K = 1$. W.l.o.g., let U be the vertex being first within the incidence list, V, W occurring after U within the stream. Also, w.l.o.g. let us assume that $[A, V\rangle$ is the first edge in the stream among the edges from $\{A, B, C\}$ to V and let $X \in \{A, B, C\}$ be the node such that $[X, W\rangle$ is the first edge among the edges from $\{A, B, C\}$ to W in the stream. We have $X_K = 1$, if the following events occur:

- A, B, C, U are chosen as $K_{3,1}$ with U being the destination node (otherwise we can not detect the edges to U).
- edges $[A, V\rangle$ and $[X, W\rangle$ are selected by the algorithm (only then we have a chance to see the remaining edges of the $K_{3,3}$ in the remainder of the stream).

The probability of the first event is $1/K_{3,1}$. Conditioned on the first event the probability to choose $\{v, w\} = \{V, W\}$ is $2/\Delta^2$: each edge $[x_1, .\rangle$ appearing after $[x_1, U\rangle$ in the stream has a probability of $1/\Delta$ to be chosen by the algorithm. We know that $[x_1, V\rangle$ and $[x_1, W\rangle$ appear after $[x_1, U\rangle$ in the stream. Therefore each of these two edges has a probability of $1/\Delta$ to be chosen. By a similar argument we have independently a probability of $1/\Delta$ to choose the edge $[x_2, V\rangle$ resp. $[x_2, W\rangle$. We select the nodes V and W if we either select $[x_1, V\rangle$ and $[x_2, W\rangle$ or $[x_1, W\rangle$ and $[x_2, V\rangle$. Therefore the probability to choose $\{v, w\} = \{V, W\}$ and w is $2/\Delta^2$.

Assume now that we have chosen A, B, C, U as the $K_{3,1}$ and the edges $[x_1, V\rangle$ and $[x_2, W\rangle$. This assumption is satisfied with probability $2/(\Delta^2 \cdot K_{3,1})$, independent of the choice of x_1 and x_2. The probability for $x_1 = A$ and $x_2 = X$ is $1/9$, since x_1 and x_2 are chosen uniformly from $\{A, B, C\}$. Therefore the probability to detect the $K_{3,3}$ and to set $X_K = 1$ is $2/(9 \cdot \Delta^2 \cdot K_{3,1})$. Since we fixed the $K_{3,3}$, the probability to detect an arbitrary $K_{3,3}$ this way is then $\frac{2 \cdot K_{3,3}}{9 \cdot \Delta^2 \cdot K_{3,1}}$.

To complete the description of the algorithm, we show how we can choose a $K_{3,1}$ uniformly from the stream in the first step of our algorithm. This is done using a similar method as choosing the length-2-paths in an algorithm to count the number of triangles in a graph[2]. We start $O(\log n)$ different instances of the algorithm below in parallel. Each instance corresponds to a guess $2^j, 1 \leq j \leq 4 \log n$ of the number of $K_{3,1}$ in the graph. We also count the number $K_{3,1} = \sum_{i=1}^{|V|} d_i \cdot (d_i - 1) \cdot (d_i - 2)/6$. In the end we can choose an instance whose guess is at most a factor 2 away from the number of $K_{3,1}$. So from now on let us assume we know the number of $K_{3,1}$ (up to a factor 2). We will extend the method UNIFORMTWOPATH from [2]. Let $f(x) = \binom{x}{3}$. The algorithm is as follows.

UNIFORM$K_{3,1}$
Select value k uniformly from the set $\{1, \ldots, K_{3,1}\}$.
For each node v in the incidence stream **do**:
If $k > 0$ **then**
Set $h \leftarrow \lceil f^{-1}(k) \rceil$
Set $k_2 \leftarrow k - f(h-1)$
Set $i \leftarrow \left\lceil \sqrt{2k_2 + \frac{1}{4}} + \frac{1}{2} \right\rceil$
Set $j \leftarrow i - \frac{i^2 - i}{2} + k_2 - 1$
Pass over the complete incidence list of node v.
If incidence list of v contains more than j edges **then**
$a \leftarrow$ the hth node in the incidence list of v
$b \leftarrow$ the ith node in the incidence list of v
$c \leftarrow$ the jth node in the incidence list of v
$u \leftarrow v$
end if
$d \leftarrow$ degree of node v
$k \leftarrow k - \frac{d \cdot (d-1) \cdot (d-2)}{6}$
end if
end do
return edges (a, u),(b, u) and (c, u)

The idea of the algorithm above is to enumerate the $K_{3,1}$'s that are incident to one node v in a way that all $K_{3,1}$ whose edges belong to the first i edges incident to v are enumerated before those using an edge which appears later in the stream. We select the kth $K_{3,1}$ with respect to that enumeration order. Given a number k we can compute a triple (h, i, j) telling us to select the h-th, i-th, and j-th edge incident to the current vertex. If the current vertex has less than j edges, we know that the node v is incident to less than k $K_{3,1}$ subgraphs. We simply subtract the number of $K_{3,1}$ choices for this vertex from k and start with our new k and the next vertex.

Equivalent to the algorithms of Section 4 we can run the algorithm $K_{3,3}$COUNTING in parallel $\Theta(\log \frac{1}{\delta} \cdot \frac{\Delta^2}{\epsilon^2 \cdot K_{3,3}})$ times. We devide the instances into $\Theta(\log \frac{1}{\delta})$ groups of size $\frac{27 \cdot \Delta^2 \cdot K_{3,1}}{\epsilon^2 \cdot 2 \cdot K_{3,3}}$, take the average result of each group and return the median of these group values. The returned value is with probability $1 - \delta$ a $(1 \pm \epsilon)$-approximation of the value $\frac{2 \cdot K_{3,3}}{9 \cdot \Delta^2 \cdot K_{3,1}}$. Since we can in parallel count the value $K_{3,1}$, a multiplication of the median with $\frac{9 \cdot \Delta^2 \cdot K_{3,1}}{2}$ will yield a $(1 \pm \epsilon)$-approximation of the number of $K_{3,3}$ with probability $1 - \delta$.

Theorem 3. *There is a 1-pass streaming algorithm which returns with probability* $1-\delta$ *a* $(1 \pm \epsilon)$*-approxi-mation of the number of* $K_{3,3}$ *subgraphs of a graph* G *when the graph is given as an incidence stream ordered by destination nodes with outdegree bounded by* Δ*. It uses* $\mathcal{O}\left(\log(|V|) \cdot \log(\frac{1}{\delta}) \cdot \frac{K_{3,1} \cdot \Delta^2}{K_{3,3} \cdot \epsilon^2}\right)$ *memory cells.* □

6 Implementation of the Data Stream Algorithms

In this section we describe several optimization steps used in the implementation of the algorithms to count $K_{3,3}$'s and to estimate the clustering coefficient.

First, we use some memory blocks in order to improve I/O efficiency. Instead of reading single edges consecutively from the hard disk, a large amount of edges is read at once and stored in a main memory block of fixed size. This also allows to distinguish between processing time and reading time.

Our algorithms consist of s instances passing in parallel over the stream of edges. Instead of feeding each edge to each of the s instances we use optimized hashing approaches. Each instance describes a set of possible edges or nodes that it is interested in. These sets of interesting edges are then inserted into a hashtable. After reading an edge our algorithm can then quickly identify the different instances which are interested in that particular edge (resp. the end nodes of the edge).

We use simple hash functions to implement the hash table. For hashtables storing edges (i.e. two indices u and v) we use hash functions h of the form $h(u,v) = rnd_1 \cdot u + rnd_2 \cdot v \mod 2s$, where rnd_1 and rnd_2 are two randomly generated numbers between one and s (sample set size). For hashtables storing single nodes we use hash functions h of the form $h(u) = rnd_1 \cdot u \mod 2s$. The size of the hash tables is $2 \cdot s$. Our hashtables do not provide theoretical guarantees on the lookup time. We nevertheless chose them because they provide very fast hashing time in all our experiments.

The implementations of the one pass algorithm to estimate the clustering coefficient and the two pass algorithm to count $K_{3,3}$'s are described in the next subsections.

6.1 One Pass Algorithm to Estimate the Clustering Coefficient

To simplify the implementation we store the incidence list of a single node in main memory. The memory consumption for this operation is negligible, since all our input instances are sparse graphs.

The hashing function is implemented as described above. We chose this particular set of hash functions because they are extremely fast and provide good approximations (see results).

6.2 Two Pass Algorithm to Count $K_{3,3}$'s

We implemented a 2-pass algorithm to count $K_{3,3}$'s. Our implementation assumes the following input encoding of a directed input graph: The file contains n lists, one for each node v. The first lines of the list specify the indegree and outdegree of v. The following lines provide a list of edges pointing to v.

The first pass serves the purpose of counting the number of $K_{3,1}$'s in the graph. This way we avoid the evaluation of the algorithm for a logarithmic number of guesses. Our algorithm can easily be extended to a one-pass algorithm using the guessing technique described in detail in Section 5, but the time would increase considerable due to the need of one run of the algorithm for each guess of number of $K_{3,1}$'s. For this reason, we decided to implement the two passes algorithm instead.

In the second pass we uniformly select s random $K_{3,1}$ subgraphs of the graph: We first sample a random index k between 1 and the number of $K_{3,1}$'s and retrieve the k-th $K_{3,1}$ from the stream using the algorithm UNIFORM$k_{3,1}$ (presented in Section 5).

After the selection of the $K_{3,1}$'s we select uniformly at random two vertices x_1, x_2 from a, b, c and two numbers k_1, k_2 in $\{1, 2, \ldots, \Delta\}$. It remains to detect the vertices w and v, and to check the existence of 6 different edges per instance. This can both be done using the hashing technique described above to speed up the detection of vertices and edges which are interesting for one of s parallel instances.

7 Computational Experiments

This section presents the computational experiments performed by running the 1-pass algorithm to estimate the clustering coefficient of a graph and the 2-pass algorithm to count the number of $K_{3,3}$'s.

The codes were written in C/C++ and compiled with the `gcc` compiler version 3.2.2, using the `-O3` optimization option. The experiments were performed on a 2.4 GHz Intel Pentium IV computer with 512 MB of RAM, running Linux. Due to space requirements, the experiments for the webgraph were performed on a 2.8 GHz Intel Pentium IV computer with 1 GB of RAM, running Linux. The implementations are available by e-mail request.

CPU times were measured with the system function `getrusage`. The time for reading the graphs is not included in the reported running times.

7.1 Datasets

The datasets are divided into three subsets. The first set contains a large `webgraph` instance. It consists of 135 million nodes and 1 billion edges and is obtained from a graph extracted in 2001 by the WebBase project at Stanford [16]. We removed the frontier nodes, i.e, the nodes that have indegree equal to one and outdegree equal to zero.

The second set of instances contains the input graphs of the experiments of [17]. The instances are:

- `actor2004` and `actor2004`: Based on the *Internet Movie Database*. In these instances, two actors (nodes) are connected if they ever stared together in a movie.
- `authors`: Based on the *Computer Science Bibliography* at the University of Trier.
- `google-2002`: Based on the 2002 Google contest.
- `itdk0304`: The network of Internet routers (nodes) and their connections (edges) collected by the *Cooperative Association for Internet Data Analysis* (CAIDA).

The third set of instances represents the link structure of Wikipedia, as of an old dump of June 13, 2004 [3]. Wikipedia is nowadays the largest online encyclopedia, available in more than 200 languages. In these graphs, each article is a node and each hyperlink between nodes is a directed arc. A graph is extracted from each language. The experiments were performed on the graphs `wikiEN`, `wikiDE`, `wikiFR`, `wikiES`, `wikiIT` and `wikiPT`, extracted from the English, German, French, Spanish, Italian and Portuguese languages.

Table 1. Basic properties of the graphs. The columns correspond to the number of nodes ($|V|$), number of arcs ($|E|$), minimum degree of a node (min), average degree of the nodes (avg), and maximum degree of a node (max).

	graph dimensions		degree		
Graph	$\|V\|$	$\|E\|$	min	avg	max
WebGraph	135,765,866	1,186,742,657	1	15.91	7,885,507
actor2004	667,609	27,581,275	1	82.63	4,605
google-2002	394,510	480,259	1	2.43	1,160
actor2002	382,219	15,038,083	1	78.69	3,96
authors	307,971	831,557	1	5.40	248
itdk0304	192,244	609,066	1	6.34	1,071
wikiEN	327,914	4,811,393	1	29.35	47,123
wikiDE	114,809	1,907,891	1	33.24	5,962
wikiFR	41,745	577,781	1	27.68	7,651
wikiES	25,684	246,316	1	19.18	2,973
wikiIT	12,758	134,342	1	21.06	1,793
wikiPT	8,071	42,083	1	10.43	2,317

Table 2. Results for estimating the cluster coefficient in digraphs, considering sample set sizes of 300, 1500, and 3000 for all graphs. For each instance and sample size, the table presents results for the estimated clustering coefficient $\widetilde{cc}$ and the corresponding running time measured in seconds (Time(s)). Moreover, we run an own implementation to calculate the exact clustering coefficient ($cc*$) on all graphs of feasible size.

Graph	s=300		s=1500		s=3000		
	$\widetilde{cc}$	Time(s)	$\widetilde{cc}$	Time(s)	$\widetilde{cc}$	Time(s)	cc*
webgraph	0.31	11,380.18	-	-	-	-	-
actor2004	0.77	41.94	0.76	381.00	0.77	817.33	0.76
google-2002	0.03	1.56	0.05	9.05	0.05	25.32	0.05
actor2002	0.74	34.11	0.75	305.27	0.78	669.48	0.78
authors	0.58	2.94	0.52	21.92	0.58	47.44	0.60
itdk0304	0.11	1.73	0.15	13.60	0.15	37.94	0.16
wikiEN	0.21	16.32	0.21	16.32	0.26	276.17	0.27
wikiDE	0.21	16.54	0.20	101.59	0.22	234.82	0.24
wikiFR	0.29	7.80	0.28	77.31	0.28	163.88	0.29
wikiES	0.23	6.07	0.24	50.69	0.28	163.88	0.27
wikiIT	0.27	8.23	0.25	49.72	0.27	113.33	0.30
wikiPT	0.40	3.68	0.36	23.53	0.38	48.76	0.37

Table 1 presents basic properties of these graphs. All of them are sparse. The dimensions vary from less than ten thousand nodes to more than 135 million nodes, and from less than 100 thousand edges to over one billion edges.

7.2 Clustering Coefficient Computation

Table 2 presents results for the one pass algorithm to estimate the clustering coefficient of a graph. For directed graphs we replaced all directed edges by undirected edges and

Table 3. Results for estimating the number of $K_{3,3}$ in a digraph, considering sample set sizes of 100,000 and 1,000,000. For each run, the table presents the number of $K_{3,3}$'s estimated by the algorithm ($\widetilde{K_{3,3}}$) and the corresponding running time measured in seconds (`Time(s)`).

Graph	s=100,000		s=1,000,000	
	$\widetilde{K_{3,3}}$	Time(s)	$\widetilde{K_{3,3}}$	Time(s)
Webgraph	7,880,146,700,948,425	218.38	7,593,595,911,823,028	253.96
	7,880,146,700,948,425	219.26	6,017,566,571,633,343	254.48
	6,447,392,755,321,438	230.89	7,808,509,003,667,075	311.46
actor2004	5138977070	4.54	5253176561	8.75
	4838452096	4.67	5481575542	9.99
	5199082066	4.75	5346339303	9.20
google-2002	8859112	1.19	4724860	5.98
	0	1.13	4134252	6.04
	8859112	1.12	5315467	6.28
actor2002	138317076	1.39	188762127	8.74
	154589673	1.32	174116790	7.47
	162725972	1.35	180625828	7.18
authors	27783122	1.40	31638738	6.79
	35154154	1.26	31978940	6.79
	27783122	1.35	30334632	6.87
itdk0304	2550135420	1.30	2370936715	9.98
	2653519288	1.33	2284783492	10.71
	2067677368	1.58	2426074778	9.94
wikiEN	0	1.86	0	8.63
	0	1.90	1663312320	8.59
	0	1.83	831656160	8.68
wikiDE	0	1.34	408172513	8.21
	1530646924	1.17	612258770	8.14
	0	1.29	510215641	8.26
wikiFR	293980344	0.85	88194104	8.32
	293980344	0.86	146990172	8.31
	0	0.83	176388206	8.21
wikiES	163764410	0.75	223811360	7.50
	327528819	0.65	234728987	7.52
	191058478	0.71	259293648	7.45
wikiIT	62773047	0.68	43941133	7.38
	41848698	0.69	43941133	7.39
	20924349	0.67	35571394	7.47
wikiPT	1582891085	0.81	1559913634	9.02
	1442473328	0.79	1529277032	8.92
	1442473328	0.80	1589273710	9.02

generated the complete adjacency list for each node. We ran the algorithm on each graph with three different sample sizes ($s = 300$ and $s = 1500$, and $s = 3000$).

Unfortunatelly we could not compare our results with the results presented by [17], since they used a slightly different definition of clustering coefficient, resulting in a $cc*$ different from our.

For most of the graphs, the data stream algorithm was able to find an estimated clustering coefficient very close to the exact one represented by cc* in the table. Most of the time was spent in the operation of searching nodes in an adjacency list (there are two such operations in the algorithm).

On the webgraph we were not able to compute the exact clustering coefficient. Having the good approximations of all other instances in mind, we expect the approximation guarantee to be around 2%.

7.3 $K_{3,3}$ Computation

Table 3 presents results for the two pass algorithm to estimate the number of $K_{3,3}$'s of a directed graph. As it is common in this kind of computation [12,14], we removed nodes having an indegree of more than 50. In [12] the authors concluded that pages with indegree higher than 50 are presumably referenced for a variety of reasons not having to do with any single emerging community. On average 15% of the nodes and 42% of the edges were removed.

Using a sample size of 1,000,000 almost all runs were able to find $K_{3,3}$'s in the sample set. The number of samples that detect a $K_{3,3}$ varies considerably among the instances. Considering a sample size of 1,000,000, hundreds and thousands of $K_{3,3}$'s were found in almost all instances. This indicates that there are strongly related communities in the respective graphs. Exceptions are some instances from the third set (some `wiki` graphs), and `google-2002`. We observe that in all instances for which a non-zero number of $K_{3,3}$'s are found in all the runs, the values reported with sample size $100,000$ are very close to those reported for sample size $1,000,000$. The variation among the results for a fixed instance and sample size is directly related to the number of $K_{3,3}$'s found in each run. In some cases, only a few $K_{3,3}$'s were found, and so the results are sensible a small variations in this number. Running times are very short in all runs, even for a sample size of 1,000,000. The hashing tricks described in Section 6 seem to lead to a very small dependency of the runtime on the sample size s.

Acknowledgements

We thank Thomas Schank and Dorothea Wagner for five of the intances they have used in their paper [17]. We also thank A. Broder and S. Muthukrishnan for helpful discussions and Debora Donato for helping to organize the webgraph files.

References

1. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments
2. Buriol, L., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., Sohler, C.: Counting triangles in data streams. In: Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 06 (2006)
3. Buriol, L.S., Castillo, C., Donato, D., Leonardi, S., Millozzi, S.: Temporal evolution of the wikigraph. In: Proceedings of Web Intelligence, IEEE Computer Society Press, Hong Kong (2006)

4. Coppersmith, D., Winograd, S.: Matrix multiplication via arith- metic progressions. Journal of Symbolic Computation 3(9), 251–280 (1990)
5. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating System Design and Implementation (2004)
6. Eubank, S., Kumar, V.S.A., Marathe, M.V., Srinivasan, A., Wang, N.: Structural and algorithmic aspects of massive social networks. In: SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 718–727. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2004)
7. Sohler, C., Frahling, G., Indyk, P.: Sampling in dynamic data streams and applications. In: 21st Annual Symposium on Computational Geometry, pp. 142–149. ACM Press, New York (2005)
8. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In: VLDB, pp79–88 (2001)
9. Guha, S., Koudas, N., Shim, K.: Data-streams and histograms. ACM Symposium on Theory of Computing, 471–475 (2001)
10. Henzinger, M., Raghavan, P., Rajagopalan, S.: Computing on data streams (1998)
11. Jowhari, H., Ghodsi, M.: New streaming algorithms for counting triangles in graphs. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 710–716. Springer, Heidelberg (2005)
12. Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.: Trawling the web for emerging cyber communities. In: Proc. of the 8th WWW Conference, pp. 403–416 (1999)
13. Kumar, R., Raghavan, P., Rajagopalan, S., Sivakumar, D., Tomkins, A., Upfal, E.: Random graph models for the web graph. In: FOCS, pp. 57–65 (2000)
14. Laura, L., Leonardi, S., Millozzi, S., Sybeyn, J.F.: Algorithms and experiments for the web-graph. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, Springer, Heidelberg (2002)
15. Muthukrishnan, S.: Computing on data streams (2005)
16. The Standord WebBase Project, http://www-diglib.stanford.edu/~testbed/doc2/webbase/
17. Schank, T., Wagner, D.: Finding, counting and listing all triangles in large graphs, an experimental study. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 606–609. Springer, Heidelberg (2005)
18. Watts, D.J., Strogatz, S.H.: Collective dynamics of small- world networks. Nature 393, 440–442 (1998)
19. Sivakumar, D., Bar-Yosseff, Z., Kumar, R.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 623–632. ACM Press, New York (2002)

Complete, Exact and Efficient Implementation for Computing the Adjacency Graph of an Arrangement of Quadrics*

Laurent Dupont[1], Michael Hemmer[2], Sylvain Petitjean[1], and Elmar Schömer[2]

[1] LORIA, Nancy, France
{laurent.dupont,sylvain.petitjean}@loria.fr
[2] Johannes Gutenberg-Universität, Institut für Informatik, Mainz, Germany
{hemmer,schoemer}@informatik.uni-mainz.de

Abstract. We present a complete, exact and efficient implementation to compute the adjacency graph of an arrangement of quadrics, *i.e.* surfaces of algebraic degree 2. This is a major step towards the computation of the full 3D arrangement. We enhanced an implementation for an exact parameterization of the intersection curves of two quadrics, such that we can compute the exact parameter value for intersection points and from that the adjacency graph of the arrangement. Our implementation is *complete* in the sense that it can handle all kinds of inputs including all degenerate ones, *i.e.* singularities or tangential intersection points. It is *exact* in that it always computes the mathematically correct result. It is *efficient* measured in running times, *i.e.* it compares favorably to the only previous implementation.

1 Introduction

Quadric surfaces are the simplest curved surfaces. They play an important role in solid modeling and in the design of mechanical parts. Unfortunately, when dealing with curved objects, including quadric surfaces, all CAD systems still suffer from approximation and rounding errors due to the use of fast but inexact floating point arithmetic. As a consequence, all of these systems are neither exact nor complete. On the other hand, computer algebra introduced very general methods, *e.g.* the cylindrical algebraic decomposition presented by Collins [1], which are exact and complete in principle but cause unacceptable runtimes.

However, in recent years, there has been substantial effort to join the three goals (exactness, completeness and efficiency) for dealing with curved objects. In particular the problem of computing the arrangement of quadric surfaces has been studied extensively. Wolpert *et al.* [2] presented an exact and efficient technique to compute a cell in an arrangement of quadric surfaces. Mourrain *et al.* [3] presented an approach for sweeping an arrangement of quadrics in 3D, but an implementation is still missing. Finally Berberich *et al.* [4] developed and

* Project co-funded by the European Commission within FP6 (2002–2006) under contract No. IST-006413.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 633–644, 2007.

implemented a method for computing the arrangement of curves on the surface of one quadric induced by all the other quadrics. By contrast with our method, it is based on the computation of the planar arrangement of the projected intersection curves. We will compare our method to this *projection approach* in Section 6.3.

We now sketch our overall algorithm for computing the adjacency graph. The first step is to compute the exact parameterization of all appearing intersection curves as it is presented by Dupont *et al.* [5], which we recall tersely in Section 2. In the next step we unify representations of identical intersection curves, as discussed in Section 3. Thereafter, for each intersection curve we compute the exact parameter values of the intersection points with the other quadrics, as discussed in Section 4. This is a major ingredient to our approach, since this gives the possibility to sort intersection points along each intersection curve. However, each intersection point has several representations, one parameter value for each curve it lies one. Therefore, it is necessary to identify the different representations of each intersection point, which is discussed in Section 5. From that it is possible to compute the adjacency graph connecting all vertices of the arrangement with their neighbors. Section 6 presents implementation details and benchmarks in particular with respect to the projection approach. The paper closes with Section 7 on conclusions and further work, *i.e.* sketching what is needed to compute the complete 3D arrangement.

2 Intersecting Two Quadrics

In this section, we recall definitions and the main results on the exact parameterization of the intersection of quadrics as presented in Dupont *et al.* [5].

2.1 Definitions and Notations

We work in projective space $\mathbb{P}^3$, the set of quadruplets $\mathbf{X} = (x_0, x_1, x_2, x_3) \neq (0, 0, 0, 0)$ with the equivalence relation $(x_0, x_1, x_2, x_3) \sim (\lambda x_0, \lambda x_1, \lambda x_2, \lambda x_3)$ for all $\lambda \neq 0$. A quadric surface is defined in $\mathbb{P}^3$ by an implicit equation of degree 2: $\sum_{0 \leqslant i \leqslant j \leqslant 3} \alpha_{ij} x_i x_j = 0$. Since this equation can also be expressed as $\mathbf{X}^T Q \mathbf{X} = 0$, with Q a 4×4 symmetric matrix, we associate the quadric to this matrix Q.

An essential notion for parameterizing intersections is the *pencil*$(Q_1, Q_2) = \{\lambda Q_1 + \mu Q_2 \mid (\lambda, \mu) \in \mathbb{P}^1\}$ of two quadric surfaces, since the intersection curve of two quadrics can be identified by the corresponding pencil and vice versa. The different possible variants of intersection curves can in a first step be separated by the root pattern of the characteristic polynomial of the pencil: $\det(\lambda Q_1 + \mu Q_2)$.

2.2 Generic Case

In this case, $\det(\lambda Q_1 + \mu Q_2)$ has only simple real roots (and so is square-free) and the intersection curve is a smooth quartic curve of genus 1. To compute the

Table 1. Maximal degree of algebraic extension for the different algebraic components. In case of extension of degree 4, the extension can be irreducible or the direct sum of 2 extensions of degree 2 (2×2). In most cases, the algebraic degree of the parameterizations computed is optimal (one cannot do better). In the remaining cases, this degree is near-optimal, *i.e.* at most one square root away from the optimal.

component	maximal algebraic degree of extension	degree for the entire intersection	optimality
regular quartic	2	2	1 possible extra square-root
nodal quartic	2	2	1 possible extra square-root
cuspidal quartic	1	1	optimal
cubic	1	1	optimal
conic	4 (2×2)	4	1 possible extra square-root
line	4	24	optimal
point	4 (2×2)	4	optimal

parameterization Dupont *et al.* [5] chose a ruled quadric $Q_R \in pencil(Q_1, Q_2)$ (assumed different from Q_1 w.l.o.g.) such that it has a parameterization[1] $\mathbf{X}_R(\xi, \tau) \in [\mathbb{Q}(\sqrt{\delta})[\xi, \tau]]^4$, with $(\xi, \tau) \in \mathbb{P}^1 \times \mathbb{P}^1$ and where $[\mathbb{Q}(\sqrt{\delta})[\xi, \tau]]^4$ is the vector space of dimension 4 of bivariate polynomials with coefficients in an algebraic extension of degree 2. Moreover, $\mathbf{X}_R$ has the nice property that every coordinate is **bilinear** in ξ and τ. Then $\mathbf{X}_R$ is plugged into the implicit equation of Q_1 to obtain the **biquadratic** implicit equation $f := {\mathbf{X}_R}^T Q_1 \mathbf{X}_R \in \mathbb{Q}(\sqrt{\delta})[\xi, \tau]$ of the intersection curve within the parameter space of $\mathbf{X}_R$:

$$f(\xi, \tau) = a_2\tau^2 + a_1\tau + a_0, \quad (a_i)_{i=0,1,2} \in \mathbb{Q}(\sqrt{\delta})[\xi]. \tag{1}$$

Since f is quadratic in τ (but also in ξ which implies that the a_i are quadratic), solving for τ yields:

$$\tau_\varepsilon(\xi) = \left(-a_1 + \varepsilon\sqrt{\Delta}\,,\, 2a_0\right), \tag{2}$$

where $\varepsilon = \pm 1$ and $\Delta = a_1^2 - 4a_0a_2 \in \mathbb{Q}(\sqrt{\delta})[\xi]$ is of degree 4. Using this within the parameterization $\mathbf{X}_R$ of Q_R, the final parameterization of the intersection curve of Q_1 and Q_2 is defined as:

$$\mathbf{X}_\varepsilon(\xi) = \mathbf{X}_R(\xi, \tau_\varepsilon(\xi)) \in [\mathbb{Q}(\sqrt{\delta})[\xi, \sqrt{\Delta}]]^4.$$

Therefore, the parameterization consists of two arcs, a positive arc $\mathbf{X}_{\varepsilon=1}$ and a negative arc $\mathbf{X}_{\varepsilon=-1}$, each defined within the domain $\mathbb{D} = \{\xi \in \mathbb{P}^1 \mid \Delta(\xi) \geqslant 0\}$.

[1] For example, the parameterization of $x_0^2 + x_1^2 - x_2^2 - \delta x_3^2 = 0$, with $\delta \in \mathbb{Q}$, is $\mathbf{X}(\xi, \tau) = \left(\xi + \tau, \xi\tau - 1, \xi - \tau, (\xi\tau + 1)/\sqrt{\delta}\right)^T$, with $(\xi, \tau) \in \mathbb{P}^1 \times \mathbb{P}^1$. Note here that for the sake of simplicity we use an "affine-like" notation, where we should strictly speak about $\xi = (u, v) \in \mathbb{P}^1$, $\tau = (s, t) \in \mathbb{P}^1$, and the exact "projective" expression of parameterization $\mathbf{X}(\xi, \tau) = \left(ut + vs, us - vt, ut - vs, (us + vt)/\sqrt{\delta}\right)^T$.

2.3 Singular Case

In case $\det(\lambda Q_1 + \mu Q_2)$ is not square-free, additional invariants (type – real or complex – and multiplicity of the roots, projective type of the associated quadrics, ...) are computed to give a complete case distinction along the different algebraic components the intersection curve might consist of. Beside the nodal quartic or cuspidal quartic, the curve might consist of combinations of lines, conics and cubics[2], such that the accumulated algebraic degree never exceeds 4.

In all singular cases Dupont *et al.* [5] give a polynomial parameterization of each algebraic component. Note that the parameterizations may be defined in an algebraic extension, an overview is given in Table 1.

3 Matching Algebraic Component

It is clear that we should represent each algebraic component only once. Therefore, we need to detect equal algebraic components, which was not discussed in [5]. Obviously, we need only to compare algebraic components of the same nature. For **quartics** (nodal, cuspidal or smooth) the situation is easy, since a quartic unambiguously defines its pencil. Hence, it suffices to compare the associated pencils. In the case of **cubics**, non-equivalent pencils may contain the same cubic. However, each pencil contains up to one cubic only. Hence, we plug the parameterization of one cubic into the pencil of the second and the cubics are equal iff the result vanishes. For **conics** we have to take into account that each associated pencil may contain up to two conics. However, in case of unequal pencils only one conic may be contained in both pencils and we follow the same approach as for the cubics. In case of equal pencils containing two conics we can guarantee an identical parameterization of the conics by construction, see Dupont *et al.* [5, Part II].

In the case of **lines**, we have to compare lines defined in an algebraic extension of similar degree only, since the degree is guaranteed to be optimal (see Table 1). In case of degree one and two, we compare the lines via Plücker Coordinates [6] using explicit arithmetic[3]. In case of a degree 4 extension we can guarantee a unique representation of the line by construction due to the fact that the pencil is uniquely defined by the line and its three algebraic conjugates.

In the case of **lines** defined in a **degree 3 extension** the situation is more complicated, since the pencil is not unique due to the fourth rational line. Moreover, we want to avoid Plücker Coordinates since this involves arithmetic[4] with algebraic numbers of degree 3. Therefore, we first use the fact that all lines in a triple intersect in a common rational point. If the rational points associated with each triple are equal, we compute for each triple the three intersection points with a rational plane. Then we compute approximations of the points using interval arithmetic with increasing precision until we detect inequality or get a

[2] It may also consist of isolated points, real part of a complex component.

[3] For instance, `leda::rational` or `NiX::Sqrt_extension` (Section 6).

[4] We could use *e.g.* `leda::real` or `CORE::Expr` here, but in case of equality the numbers have to be refined until the separation bound is reached, which is too expensive.

one-to-one matching among the points of the triples, see also Section 5. In order to detect equality, we project the intersection curves of the two pencils along a certain direction. Each resultant is a bivariate polynomial of degree 4, representing the projection of the four lines in the respective pencil. Since we know the rational line of each pencil, we can divide each resultant by the respective linear factor and compare the primitive parts of the remaining polynomials. Note that we still can not state equality of the triples, since the lines could still differ due to a scaling along the chosen direction. Therefore, we have to repeat this process along a different, linear independent direction.

4 Intersection with a Third Quadric

When the intersection $C_{12} = Q_1 \cap Q_2$ of the first two quadrics is **singular**, the computation of the intersection of C_{12} with a third quadric Q_3 is straightforward. Indeed, in this case, each algebraic component of C_{12} has a polynomial parameterization $\mathbf{X}(\xi)$ of minimal degree. Therefore, it is enough to plug $\mathbf{X}(\xi)$ into the equation of the third quadric and solve the resulting univariate polynomial $h(\xi) = \mathbf{X}(\xi)^T Q_3 \mathbf{X}(\xi)$. Since $\mathbf{X}(\xi)$ is a proper parameterization of minimal degree, h has minimal degree and its real roots are in one-to-one correspondence with the intersection points of $Q_1 \cap Q_2 \cap Q_3$ on the given component of C_{12}.

The situation is slightly more involved when the intersection is **generic**, *i.e.* a smooth quartic, due to the fact that the parameterization is no longer polynomial. We focus on this case in this section.

4.1 Intersecting a Smooth Quartic with a Third Quadric

Recall that for a **smooth quartic** $C_{12} = Q_1 \cap Q_2$ the parameterization splits into two arcs, $\mathbf{X}_{\varepsilon=1}$ and $\mathbf{X}_{\varepsilon=-1}$. To compute its intersection points with a third quadric Q_3, we switch to the parameter space of the chosen ruled quadric $Q_R \in pencil(Q_1, Q_2)$. Stated simply, the algorithm below is based on the fact that

$$Q_1 \cap Q_2 \cap Q_3 = Q_1 \cap Q_R \cap Q_3 = (Q_R \cap Q_1) \cap (Q_R \cap Q_3).$$

Let again $f(\xi, \tau)$ be the implicit equation of C_{12} as defined in (1) and $\tau_\varepsilon(\xi)$ be its parameterization as defined in (2). In symmetry to f, we denote the implicit equation representing $Q_R \cap Q_3$ by:

$$g(\xi, \tau) = \mathbf{X}_R{}^T Q_3 \mathbf{X}_R = b_2\tau^2 + b_1\tau + b_0, \quad (b_i)_{i=0,1,2} \in \mathbb{Q}(\sqrt{\delta})[\xi]. \tag{3}$$

Moreover, we denote $resultant_\tau(f, g)$ by:

$$res = s_{02}^2 - s_{01}s_{12}, \quad s_{ij} = a_i b_j - a_j b_i \ \in \mathbb{Q}(\sqrt{\delta})[\xi]. \tag{4}$$

If $Q_R \cap Q_1 = Q_R \cap Q_3$, res is the zero polynomial. Otherwise, res is of degree 8 and its roots are the parameter values of the 8 intersection points of the three quadrics, multiplicities counted. However, it remains to discard the complex intersection points and to determine the correct arc for the real intersection points. This is embodied in Theorem 1.

Theorem 1 (Description of Points on Arcs). *Let f, τ_ε, g and $res \neq 0$ be defined as in Equations (1), (2), (3) and (4) respectively. And let ξ_0 denote one of the real roots (if any) of res.*

There are 3 cases:

1. *$\Delta(\xi_0) < 0$: ξ_0 corresponds to complex intersection points.*
2. *$\Delta(\xi_0) = 0$: ξ_0 corresponds to a real endpoint of both arcs.*
3. *$\Delta(\xi_0) > 0$:*
   - *If $s_{01}(\xi_0) \neq 0$, then ξ_0 corresponds to one real intersection point on arc $\mathbf{X}_\varepsilon(\xi)$, with $\varepsilon = -\mathbf{sign}(s_{01})\mathbf{sign}(2a_0s_{02} - a_1s_{01})|_{\xi_0}$.*
   - *If $s_{01}(\xi_0) = 0$, then ξ_0 corresponds to two real points, one on each arc.*

Proof. Since the two first statements are trivial we focus on the third. When $\Delta(\xi_0) > 0$ it should be clear the intersection has at least one real point, since Δ is the discriminant in τ of the bi-degree $(2, 2)$ equation f. Now, the most important observation is that $g(\xi, \tau_\varepsilon(\xi))$ can be written as:

$$g(\xi, \tau_\varepsilon(\xi)) = 2(2a_0s_{02} - a_1s_{01} + \varepsilon \cdot s_{01}\sqrt{\Delta}), \tag{5}$$

and that this expression is independent of ε iff $s_{01}(\xi_0) = 0$. Thus if $s_{01}(\xi_0) \neq 0$ we just solve for ε. Otherwise it is clear that both choices for ε yield a valid solution. □

Remark 1. It is easy to see that it is possible to determine the multiplicities of the intersection points by investigating derivatives of Equation (5). However, since this result is not needed for our algorithm it has been omitted in Theorem 1.

4.2 Intersection Algorithm

We first compute the possible ξ values and their multiplicity by Algorithm 1.

Algorithm 1. Let C_{12} be the smooth quartic intersection curve of the quadrics Q_1 and Q_2, let $f(\xi, \tau)$ be the corresponding implicit equation within the parameter space of $Q_R \in pencil(Q_1, Q_2)$. Compute possible parameter values of intersection points with a third quadric $Q_3 \notin pencil(Q_1, Q_2)$.

$g(\xi, \tau) := \mathbf{X}_R{}^T Q_3 \mathbf{X}_R$ $\{Q_R \cap Q_3\}$
$res(\xi) := resultant_\tau(f, g)$ {of degree 8}
{compute the square-free factorization of res, by Yun's algorithm [7]}
$Sqf(f) := \{(fac_0, m_0), \ldots, (fac_k, m_k)\}$ with $\prod_{i=0}^{k} fac_i^{m_i} = res$
for all $fac_i \in Sqf(f)$ **do**
  compute all real roots of fac_i using the Bit-Stream Descartes algorithm [8]
  output all real roots of fac_i with their corresponding multiplicity m_i
end for

In the next step we compute the arc(s) to which each pair (ξ_j, m_j) belongs according to Theorem 1. Note that we only discuss $\Delta(\xi_j) > 0$ and omit the

other cases. If $s_{01}(\xi_j) \neq 0$,[5] we can evaluate $-\mathbf{sign}(s_{01})\mathbf{sign}(2a_0s_{02} - a_1s_{01})$ at ξ_j using interval arithmetic with increasing precision until we can report an unambiguous sign, and therefore assign the value to the correct arc. Otherwise, *i.e.* $s_{01}(\xi_j) = 0$, we report both arcs.

5 Matching Intersection Points

We will now focus on how to compare two intersection points, as they have been defined in Section 4. This problem actually consists of two parts. The first part is to compare and sort points along a curve, but this reduces to comparing the exact parameter values using `Algebraic_real`, see Section 6.

The second part comes from the fact that an intersection point lies on at least three quadrics. These three quadrics induce three intersection curves each going through that intersection point. Thus each intersection point has at least three representations, one for each algebraic component it lies on, and the task is to match these representations efficiently. Our solution is presented in Algorithm 2.

Algorithm 2. Let C_{12} and C_{13} be the intersection curves $Q_1 \cap Q_2$ and $Q_1 \cap Q_3$, respectively. Moreover, let $seq_{12} = \{p_{12}^1, \ldots, p_{12}^m\}$ and $seq_{13} = \{p_{13}^1, \ldots, p_{13}^m\}$ be the sequence of intersection points on the curves C_{12} (representing $C_{12} \cap Q_3$) and C_{13} (representing $C_{13} \cap Q_2$), respectively, where m is the total number of points. Compute the permutation σ that matches the points in seq_{12} with those in seq_{13}.

precision := 53 {start with double precision}
$\sigma := \{\}$ {start with the empty permutation}
while seq_{12} and seq_{13} are not empty **do**
  for all intersection points left in the game **do**
    pre-compute a bounding box using interval arithmetic with the given precision
  end for
  if a $BBOX(p_{12}^i)$ intersects one $BBOX(p_{13}^j)$ only **then**
    add (i, j) to σ
    remove p_{12}^i from seq_{12} and p_{13}^j from seq_{13}
  end if
  precision $*=$ 2 {double the precision}
end while
return σ {termination is guaranteed due to the increasing precision and the fact that each point appears exactly once in each sequence}

6 Implementation and Benchmarks

The software consists of two parts. The first part is an adaptor to the software QI [9] which provides the exact parameterization of an intersection curve of two quadrics. The second part, which is the focus of this paper, provides the

[5] Note that having the multiplicity m_j of the parameter values helps to avoid unnecessary exact calculations, *i.e.* $m_j = 1$ implies $s_{01}(\xi_j) \neq 0$.

computation of the intersection of such a curve with a third quadric. It is implemented within the EXACUS project [10]. The design of EXACUS follows the generic programming paradigm with C++ templates similar to well-established design principles, for example, in the STL [11] and in CGAL [12]. We can parameterize our implementations on such a small scale as number types and arithmetic operations. For more on content, design, and implementation of EXACUS see [10].

6.1 Implementation Details

In order to give a certain background for the benchmarks in Sec. 6.3 we now discuss the most important types and methods used within our implementation.

Sqrt_extension: As discussed in Section 2 the parameterization of an intersection curve is in general defined over an algebraic extension. As a consequence, all exact arithmetic operations are carried out within these extensions.

In the rare case of algebraically conjugate lines in which the algebraic extension is of degree up to 4, we use `leda::real` or `CORE::Expr`. For the other cases, we introduced a special class template `Sqrt_extension<NT,ROOT>`. An object of this type stores two coefficients `a1` and `a2` of type `NT` and the extension `root` of type `ROOT` representing the value $a_1 + a_2\sqrt{root}$. Note that both `NT` and `ROOT` can themselves be an instance of `Sqrt_extension`, yielding a nested extension, which is needed in the case of conics.

The type has been designed such that only values from the same extensions are interoperable. From there, it is possible to keep the representation of objects very simple. In particular, it avoids the overhead of other more general implementations based on expression trees, such as `leda::real` or `CORE::Expr`, *i.e.* it can be used as coefficient type within gcd computations and allows the use of modular filters, as discussed in [13].

Square-Free Factorization: For polynomials over integer coefficients we use Yun's algorithm [7] to compute the square-free factorization. For gcd computations we use the subresultant algorithm as presented in [14]. For polynomials over an algebraic extension of degree 2 the situation significantly changes. While each irreducible polynomial in $\mathbb{Z}[x]$ is still irreducible in $\mathbb{Q}[x]$, this is not true for polynomials in $\mathbb{Z}[\sqrt{\delta}][x]$ with respect to $\mathbb{Q}[\sqrt{\delta}][x]$.[6] Therefore, we modified Yun's algorithm, such that it calls `gcd_utcf`, a function that computes the gcd *up to a constant factor* in $\mathbb{Z}[\sqrt{\delta}][x]$.

Root Isolation: For each square-free factor we compute isolating intervals. For polynomials in an algebraic extension of degree 2 we use a method presented by Eigenwillig *et al.* [8]: the coefficients of the polynomial are converted to (potentially infinite) bit-streams and a variant of the well-known Descartes method (*i.e.*, a method based on Descartes' Rule of Signs, see [15]) is used to determine the isolating intervals of the real roots. This method performs far better than the original Descartes method on polynomials in $\mathbb{Z}[\sqrt{\delta}][x]$ since the bit-stream

[6] As an example take: $2x^2 + 2\sqrt{2}x + 1$, which is irreducible in $\mathbb{Z}[\sqrt{2}][x]$, but factors into $1/2(2x + \sqrt{2})^2$ in $\mathbb{Q}[\sqrt{2}][x]$.

approach reduces the overhead caused by the square root. For polynomials in $\mathbb{Z}[x]$ we use the original Descartes Method [15].

Algebraic_real: This class has been introduced to represent x-coordinates within planar arrangement computations of algebraic curves. Therefore, it has been designed to provide efficient comparisons, but arithmetic operations are not supported. The class template `Algebraic_real` stores an algebraic number in the form of a square-free polynomial and an interval isolating exactly one root. Template parameters are the coefficient type of the polynomial and the boundary type of the isolating interval. For the approach presented in this paper it has been revised in order to support `Sqrt_extension` as coefficient type as well. For a more detailed discussion, in particular on general optimizations, see [13].

6.2 The Projection Approach

We next discuss the projection approach by Berberich *et al.* [4] briefly, since we will compare to it in Section 6.3. The idea is to split the computation of the 3D arrangement into two steps. In the first step, for each quadric Q_i of a given set $\{Q_1, \ldots, Q_n\}$, the 2D arrangement induced by the other quadrics on the surface of Q_i is computed. In the second step, the plan is to use these arrangements in order to deduce the complete 3D arrangement. Although only the first step is presented in Berberich *et al.* [4], we consider this as a very promising approach.

For each quadric Q_i, two planar arrangements are computed, one for the lower and one for the upper part of Q_i. All intersection curves $Q_i \cap Q_j, j \neq i$ are projected onto the xy-plane by a resultant computation, which results in curves of algebraic degree 4. Then each curve is split into arcs with respect to the lower and upper part of Q_i. Thereafter, the two planar arrangements are computed separately by a variant of the Bentley-Ottmann sweep-line algorithm [16].

Although the projected curves are of algebraic degree 4, the coordinates of the event points are of algebraic degree 8 only, since they can be deduced by a multi-resultant computation using the three involved quadrics. Thus in general the basic operations, such as root isolation or gcd computation, are performed over univariate polynomials of degree up to 8 with integer coefficients. The main disadvantage of the approach is that in case of covertical events on the same curve a shear of the coordinate system must take place and everything has to be recomputed with respect to the new coordinate system. Moreover, it does not provide an explicit parameterization of the appearing intersection curves.

6.3 Benchmarks

We have not analyzed the worst case in the bit-complexity model, since we argue that our algorithms are adaptive in the bit-complexity and a worst-case analysis would not be representative. Instead, we want to show that our parameterization method is practical, *i.e.* efficient, for computing arrangements. In particular we want to analyze the competitiveness of our approach with respect to the projection approach. Since both approaches were implemented within the Exacus project, we were able to benchmark them in parallel.

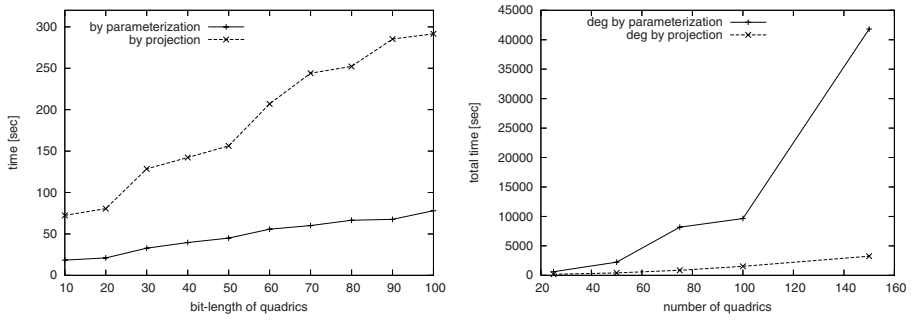


Fig. 1. To the left, parameterization approach compared to the projection approach on the random instances (rnd). To the right, parameterization approach compared to the projection approach on the degenerate instances (deg).

To analyze the generic, smooth quartic case, we generated random instances (rnd) with growing coefficient size. Each instance contains 50 quadrics with random coefficients of the respective length. In order to avoid empty quadrics or intersections, each quadric is guaranteed to intersect the $[-100, 100]^3$ cube at least once. The second set with degenerated instances (deg) is taken from Berberich *et al.* [4]. These instances have been generated by interpolation. In order to achieve degenerate situations, several quadrics share values for partial derivatives and higher-order derivatives at common intersection points. The coefficients of the set need 73 bits on average.

For the projection approach the two planar arrangements on the first quadric are computed. To be symmetric, we therefore compute all intersection curves and intersection points on the first quadric only. Moreover, we sort all points along their curve and match them with their corresponding representations on the other curves. The benchmarks were measured on a Pentium(R) M processor 1.7 GHz with 512 kB cache under Linux and the GNU C++ compiler v4.1 with optimizations (`-O2`) and disabled assertions (`-DNDEBUG`).

The right plot in Figure 1 shows that for the degenerate instances the projection approach performs better than the parameterization approach. This is due to the fact that for the degenerate situations it is not possible to avoid the computation of very expensive gcds, as is the case for equal algebraic numbers (*e.g.* equal intersection points) or non-square-free resultants (*e.g.* tangential intersection points). These computations can not be avoided by both approaches. However, in our case this is much more expensive for two reasons. First, the intersection curves are represented in the parameter space of the chosen quadric Q_R, which introduces a considerable amount of extra bits into the representation, *e.g.*, the coefficient size in the resultant polynomial for 50 bit-quadrics is around 4,500 bits while the size of the corresponding polynomial in the projection approach is only around 1,000 bits. Second, we can not apply the more efficient algorithms for square-free factorization and gcd computation used in the projection approach, since nearly all of our polynomials are defined over an algebraic extension of degree 2. This hinders the performance of our method

greatly. Indeed, for the degenerate instances, the time for the gcd computation contributes about 80% to the total runtime of the parameterization approach.

However, for the random instances the left plot in Figure 1 shows that the parameterization approach is significantly faster than the projection approach. This shows that, for the generic case, we have been able to widely decouple our approach from the bit-size of the input.

7 Conclusions and Future Work

Both approaches to the quadrics arrangement problem have their own strengths and weaknesses. Due to the use of interval arithmetic and the bit-stream algorithm our approach is fast in the generic case. On the other hand, it introduces more bits, which makes it slower in degenerate situations. However, in the beginning we had to face the same picture for the generic cases as well. The improvement was caused by the introduction of the bit-stream algorithm within the root isolation process, which gave a tremendous speed up for our approach, while for the projection approach the runtime stagnated. In the same way, the consistent use of interval arithmetic within our approach resulted in a significant speed up. We expect, that doing the same for the projection approach is considerably harder, since there is no parameterization available. Moreover, we plan to introduce modular gcd algorithms into EXACUS, as they are presented *e.g.* in [17,18]. The use of modular arithmetic is promising and should result, in degenerate cases, in a significant speed up of gcd computations on polynomials over one-root number fields. We hope to improve our approach for these cases as much as we did for the generic case.

We are confident that our approach will be finally ahead of the projection approach, with the additional advantage of exact parameterizations of the appearing intersection curves.

With respect to further work it remains to compute the full 3D arrangement. Based on the adjacency graph of the quadric arrangement, the respective algorithm will explore the local neighborhood of each intersection point to build the faces and cells of the arrangement. This could be done by computing the arrangement of conics on a sufficiently small cube around each vertex. The arrangement itself could be stored in a variant of a structure used to represent the so-called *Selective Nef Complex* as presented in [19]. This structure is a vertex oriented structure that stores the local neighborhood around each vertex in a so-called *Sphere Map*.

References

1. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Proc. 2nd GI Conf. on Automata Theory and Formal Languages. LNCS, vol. 6, pp. 134–183. Springer, Berlin (1975) Reprinted with corrections. in: Caviness, B.F., Johnson, J.R. (eds.) Quantifier Elimination and Cylindrical Algebraic Decomposition, pp. 85–121. Springer, Heidelberg (1998)

2. Schömer, E., Wolpert, N.: An exact and efficient approach for computing a cell in an arrangement of quadrics (Special Issue on Robust Geometric Applications and their Implementations). Computational Geometry: Theory and Applications 33, 65–97 (2006)
3. Mourrain, B., Técourt, J.-P., Teillaud, M.: Sweeping of an arrangement of quadrics in 3D (Special issue, 19th European Workshop on Computational Geometry). Computational Geometry: Theory and Applications 30, 145–164 (2005)
4. Berberich, E., Hemmer, M., Kettner, L., Schömer, E., Wolpert, N.: An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In: SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry, pp. 99–106. ACM Press, New York, USA (2005)
5. Dupont, L., Lazard, D., Lazard, S., Petitjean, S.: Near-optimal parameterization of the intersection of quadrics, parts I+II+III (accepted). J. Symbolic Computation (2007)
6. Stolfi, J.: Oriented Projective Geometry: A Framework for Geometric Computations. Academic Press, New York (1991)
7. Yun, D.Y.Y.: On square-free decomposition algorithms. In: SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation, pp. 26–35. ACM Press, New York, USA (1976)
8. Eigenwillig, A., Kettner, L., Krandick, W., Mehlhorn, K., Schmitt, S., Wolpert, N.: A descartes algorithm for polynomials with bit-stream coefficients. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2005. LNCS, vol. 3718, pp. 138–149. Springer, Heidelberg (2005)
9. Lazard, S., Pearanda, L.M., Petitjean, S.: Intersecting quadrics: An efficient and exact implementation. In: ACM Symposium on Computational Geometry - SoCG'2004, Brooklyn, NY, ACM Press, New York (2004)
10. Berberich, E., Eigenwillig, A., Hemmer, M., Hert, S., Kettner, L., Mehlhorn, K., Reichel, J., Schmitt, S., Schömer, E., Wolpert, N.: EXACUS: Efficient and exact algorithms for curves and surfaces. In: 13th Annual European Symposium on Algorithms, pp. 155–166. Springer, Heidelberg (2005)
11. Austern, M.H.: Generic Programming and the STL. Addison-Wesley, Reading (1998)
12. Brönnimann, H., Kettner, L., Schirra, S., Veltkamp, R.: Applications of the generic programming paradigm in the design of CGAL. In: Jazayeri, M., Musser, D.R., Loos, R.G.K. (eds.) Generic Programming. LNCS, vol. 1766, pp. 206–217. Springer, Heidelberg (2000)
13. Hemmer, M., Kettner, L., Schömer, E.: Effects of a modular filter on geometric applications. Technical Report ECG-TR-363111-01, MPI Saarbrücken (2004)
14. Brown, W.S.: The subresultant PRS algorithm. ACM Trans. Math. Softw. 4(3), 237–249 (1978)
15. Collins, G.E., Akritas, A.-G.: Polynomial real root isolation using Descartes' rule of sign. In: SYMSAC, pp. 272–275 (1976)
16. Bentley, J.L., Ottmann, T.A.: Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput. C-28(9), 643–647 (1979)
17. Brown, W.S.: On euclid's algorithm and the computation of polynomial greatest common divisors. J. ACM 18, 478–504 (1971)
18. Langemyr, S.M.L.: The computation of polynomial GCD's over an algebraic number field. J. Symbolic Computation 8, 429–448 (1989)
19. Granados, M., Hachenberger, P., Hert, S., Kettner, L., Mehlhorn, K., Seel, M.: Boolean operations on 3D selective nef complexes: Data structure, algorithms, and implementation. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 654–666. Springer, Heidelberg (2003)

Sweeping and Maintaining Two-Dimensional Arrangements on Surfaces: A First Step*

Eric Berberich[1], Efi Fogel[2], Dan Halperin[2], Kurt Mehlhorn[1], and Ron Wein[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
{eric,mehlhorn}@mpi-inf.mpg.de
[2] School of Computer Science, Tel-Aviv University, Israel
{efif,danha,wein}@tau.ac.il

Abstract. We introduce a general framework for sweeping a set of curves embedded on a two-dimensional parametric surface. We can handle planes, cylinders, spheres, tori, and surfaces homeomorphic to them. A major goal of our work is to maximize code reuse by generalizing the prevalent sweep-line paradigm and its implementation so that it can be employed on a large class of surfaces and curves embedded on them. We have realized our approach as a prototypical CGAL package. We present experimental results for two concrete adaptations of the framework: (i) arrangements of arcs of great circles embedded on a sphere, and (ii) arrangements of intersection curves between quadric surfaces embedded on a quadric.

1 Introduction

We are given a surface S in $\mathbb{R}^3$ and a set $\mathcal{C}$ of curves embedded on this surface. The curves divide S into a finite number of cells of dimension 0 (*vertices*), 1 (*edges*) and 2 (*faces*). This subdivision is the *arrangement* $\mathcal{A}(\mathcal{C})$ induced by $\mathcal{C}$ on the surface S. Arrangements are widely used in computational geometry and have many theoretical and practical applications; see, e.g., [1,9,10].

CGAL (`http://www.cgal.org`), the Computational Geometry Algorithms Library aims at a generic and robust, yet efficient, implementation of widely used geometric data structures and algorithms. Until recently, the CGAL arrangement package has dealt only with bounded curves in the xy-plane. This forced users to clip unbounded curves before inserting them into the arrangement; it was the user's responsibility to clip without loss of information. However, this solution is generally inconvenient and outright insufficient for some applications. For example, representing the *minimization diagram* defined by the lower envelope of *unbounded* surfaces in $\mathbb{R}^3$ [13] generally requires more than one unbounded face, whereas an arrangement of bounded clipped curves contains a *single* unbounded face.

* This work has been supported in part by the IST Programme of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes), by the Israel Science Foundation (grant no. 236/06), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 645–656, 2007.

Using the algorithmic principles described in this paper, we have enhanced the arrangement package included in the latest release of CGAL (Version 3.3) to handle unbounded planar curves as well. A more generic version of the package, which handles arrangements embedded on surfaces, is implemented as a prototypical CGAL package.

Related Work: Effective algorithms for manipulating arrangements of curves have been a topic of considerable interest in recent years with an emphasis on exactness and efficiency of implementation [9]. Mehlhorn and Seel [12] propose a general framework for extending the sweep-line algorithm to handle unbounded curves; however, their implementation can only handle lines in the plane. Arrangements on spheres are covered by Andrade and Stolfi [2], Halperin and Shelton [11], and recently Cazals and Loriot [7]. Cazals and Loriot have developed a software package that can sweep over a sphere and compute exact arrangements of circles on it. Fogel and Halperin [8] exchanged the single arrangement of arcs of great circles on the sphere with six arrangements of linear segments in the plane that correspond to the six faces of a cube circumscribing the sphere. Their approach requires stitching arrangements of adjacent faces. Our approach avoids this overhead. Berberich *et al.* [6] construct arrangements of quadrics by considering the planar arrangements of their projected intersection curves. Their approach requires a postprocessing step (which, however, is not implemented), while our approach avoids the need for a postprocessing step.

Paper Outline: In Section 2 we review the Bentley–Ottmann sweep and its implementation in CGAL. We generalize the algorithm in Section 3 to a class of parametric surfaces. After describing the theoretical framework we discuss implementation details in Section 4: in particular, how to encapsulate the topology of the surface into a so-called *topology-traits class*. In Section 5 we give experimental results, and present future-work directions in Section 6.

2 The Bentley–Ottmann Sweep

Recall that the main idea behind the Bentley–Ottmann sweep-line algorithm [3] (and its generalization in [14]) is to sweep over the plane containing a set of bounded curves with a vertical line starting from $x = -\infty$ toward $x = +\infty$, while maintaining the set of x-monotone curves that intersect this line. These curves are ordered according to the y-coordinate of their intersection with the vertical line and stored in a balanced search tree named the *status structure*. The content of the status structure changes only at a finite number of *events*. The event points are sorted in ascending xy-lexicographic order and stored in an *event queue*. This event queue is initialized with all curve endpoints, and is updated dynamically during the sweep process as new intersection points are discovered. The main sweep process involves the handling of events, namely the insertion of a new curve into the status structure, the removal of a curve that reaches its endpoint, or maintaining the order of intersecting curves. In either case, curves that become adjacent in the status structure are checked for intersections to the

right of the sweep line (having lexicographically greater coordinates) and any such intersection is inserted into the event structure. For our generalization, we call this the main *sweep procedure*. It is preceded by a *preprocessing step* that subdivides, if necessary, the input curves into x-monotone subcurves.

The CGAL implementation of the sweep procedure is generic and independent of the type of curves it handles. All steps of the algorithm are controlled by a small number of geometric primitives, such as comparing two points in xy-lexicographic order, computing intersection points, etc. These primitives are encapsulated in a so-called *geometry-traits class*; see [15] for the full documentation of this concept. Different geometry-traits classes are provided in the arrangement package to handle various families of curves, such as line segments, conic arcs, etc.

The *canonical output* of the sweep-line algorithm consists of the order in which events are processed along with their adjacency information (which events are connected by a curve segment). The implementation in CGAL's arrangement package elegantly decouples the "bare sweep" procedure from the construction of the actual output using *sweep-line visitors* [16]. For example, we use a visitor to convert the canonical output of the sweep procedure into a doubly-connected edge-list (DCEL) representing the arrangement induced by the set of input curves.

3 Sweeping Surfaces

We generalize the sweep to surfaces such as half-planes, cylinders, spheres, tori, etc. and surfaces homeomorphic to these. We describe our generalization in two steps, aiming for an implementation that can handle all these surfaces with maximal code reuse. We capture the geometry of our surfaces through parameterization. A parameterized surface S is defined by a function $f_S : \mathbb{P} \to \mathbb{R}^3$, where $\mathbb{P} = U \times V$ is a rectangular two-dimensional parameter space and f_S is a continuous function. We allow $U = [u_{\min}, u_{\max}]$, $U = [u_{\min}, +\infty)$, $U = (-\infty, u_{\max}]$, or $U = (-\infty, +\infty)$, and similarly for V. Intervals that are open at finite endpoints bring no additional power and we therefore do not discuss them here. For example, the standard planar sweep corresponds to $U = V = (-\infty, +\infty)$, and $f_S(u, v) = (u, v, 0)$. The unit sphere is parameterized via $\mathbb{P} = [-\pi, \pi] \times [-\frac{\pi}{2}, \frac{\pi}{2}]$ and $f_S(u, v) = (\cos u \cos v, \sin u \cos v, \sin v)$.

A *curve* is defined as a function $\gamma : D \to \mathbb{P}$ with (i) D is an open, half-open, or closed interval with endpoints 0 and 1; (ii) γ is continuous and injective, except for *closed curves* where $\gamma(0) = \gamma(1)$; (iii) if $0 \notin D$, the arc has no start point, and emanates "from infinity". It holds $\lim_{t \to 0+} \|\gamma(t)\| = \infty$ (we have a similar condition if $1 \notin D$) and we assume that these limits exist. A *sweepable curve* must, in addition, be weakly u-monotone — that is, if $t_1 < t_2$ then $\gamma(t_1)$ precedes $\gamma(t_2)$ in lexicographic uv-ordering. Consider for example the equator curve on the sphere as parameterized above. This curve is given by $\gamma(t) = (\pi(2t-1), 0)$, for $t \in [0, 1]$.

We do not assume that either surfaces or curves are given through their parameterization. We use the language of parameterization in our definitions. The algorithm can learn about curves and points only through a well-defined set of geometric predicates provided by the geometry-traits class.

In terms of the standard sweep algorithm, it is convenient to view our sweep as taking place in the parameter space, where we sweep a vertical line from $u_{\min}$ to $u_{\max}$. This is equivalent to sweeping a curve (the image of the vertical line $u = u_0$ under f_S) over the input surface S. Typically, the desired output is embedded on S (e.g., the arrangement induced on S) so one may find it more convenient to view the sweep as taking place over S.

3.1 Bijective Parameterizations and Boundary Events

Recall that the standard Bentley–Ottmann sweep starts by a preprocessing step, which subdivides all input curves into x-monotone subcurves, and initializes the event queue with all their endpoints. When generalizing the algorithm for unbounded curves, we face the problem that these curves do not have finite endpoints. We overcome this difficulty by extending the definition of a *curve-end*, so it may either be a finite endpoint, or represent an unbounded entity in case of a curve that approaches a rim in the parameter space with one of its ends. We say that the curve-end $\langle \gamma, 0 \rangle$ *approaches the west (east) rim* if $\lim_{t \to 0+} \gamma(t) = (-\infty, v_0)$ ($(+\infty, v_0)$, respectively), for some $v_0 \in \mathbb{R} \cup \{-\infty, +\infty\}$, and that it *approaches the south (north) rim* if $\lim_{t \to 0+} \gamma(t) = (u_0, \pm\infty)$ for some $u_0 \in \mathbb{R}$. Thus, in the processing step we associate an event with the two ends of each u-monotone curve $\gamma : D \to \mathbb{P}$. The first (second) event is associated with a finite endpoint if $0 \in D$ ($1 \in D$), and with an unbounded curve-end $\langle \gamma, 0 \rangle$ ($\langle \gamma, 1 \rangle$) if $0 \notin D$ ($1 \notin D$).

In the standard sweep-line procedure, the order of events in the event queue is defined by the xy-lexicographic order of the event points. Here we augment the comparison procedure for two events to handle those events associated with unbounded curve-ends as well. This is done by subdividing the procedure into separate cases, most of which can be handled in a straightforward manner. For example, it is clear that an event approaching the "west rim" is smaller than any event associated with a finite point. When we compare two curve-ends approaching the west rim, we consider the intersection of relevant curves with a vertical line $u = u_0$ for small enough u_0 and return the v-order of these points ("small enough" means that the result does not depend on the choice of u_0). Analogous rules apply to the other situations; see Figure 1(a) for an illustration and [15] for the documentation of the full concept. Note that the rest of the sweep process remains unchanged. In Section 4 we describe how to construct a Dcel that represents an arrangement of unbounded curves.

3.2 Removing Non-injectivity on the Boundaries

So far we have discussed a simple plane-sweep, with f_S being the identity mapping. When considering more general surfaces, we must handle situations where the parameterization f_S is not necessarily bijective, so some points in S may have multiple pre-images. If we consider the example of the sphere given in the beginning of this section, then $f_S(-\pi, v) = f_S(+\pi, v)$ for all v, while $f_S(u, -\frac{\pi}{2}) = (0, 0, -1)$ and $f_S(u, \frac{\pi}{2}) = (0, 0, 1)$ for all u. The function

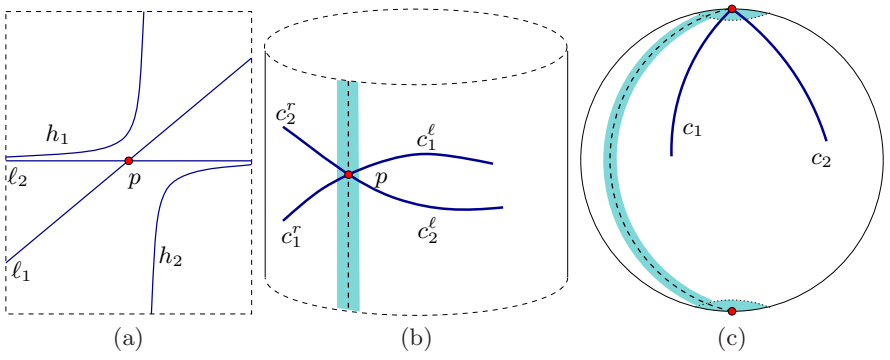
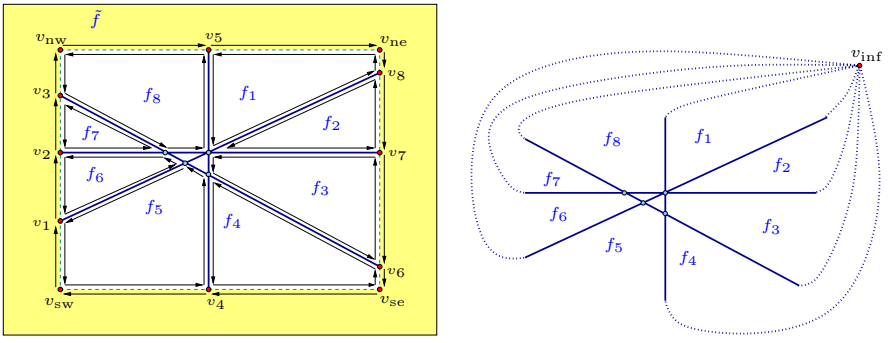


Fig. 1. Comparing sweep events: (a) The order of the events is: left end of ℓ_1, left end of ℓ_2, left end of h_1, right end of h_1, intersection of ℓ_1 and ℓ_2, left end of h_2, right end of h_2, right end of ℓ_2, and right end of ℓ_1. (b) Comparing near the curve of identification: $c_2^\ell < c_1^\ell$ right of the point p and $c_2^r > c_1^r$ to the left of p. (c) Comparing near a contraction point: $c_1 < c_2$.

$v \mapsto f_S(-\pi, v)$ is a meridian on the sphere, analogous to the "international date line", and the points $(0, 0, \pm 1)$ are the south and the north poles, respectively. The "date line" is induced by the non-injectivity of f_S: as a result, a closed curve on the surface, such as the equator on the sphere, is the image of a non-closed curve in parameter space. The poles also pose a problem: observe that they lie on the sweep line during the entire sweep.

In order to model cylinders, tori, spheres, paraboloids, and surfaces homeomorphic to them, we relax the requirements for the surface parameterizations. First, we require bijectivity only in the *interior* of $\mathbb{P}$ and allow non-injectivity on the boundary (denoted $\partial\mathbb{P}$). More precisely, we require that $f_S(u_1, v_1) = f_S(u_2, v_2)$ and $(u_1, v_1) \neq (u_2, v_2)$ imply $(u_1, v_1) \in \partial\mathbb{P}$ and $(u_2, v_2) \in \partial\mathbb{P}$. On the boundary, we allow injectivity in a controlled way:

- *Contraction* of a side of the parameter space is possible if this side is closed. For example, if $u_{\min} \in U$ and the west rim is contracted, we have $\forall v \in V, f_S(u_{\min}, v) = p_0$ for some fixed point $p_0 \in \mathbb{R}^3$, so the entire west rim is mapped to the same point p_0 on S. We call such a point a *contraction point*. For the sphere, we have contraction at the south and north rims of the parameter space, inducing the south and north poles, respectively.
- *Identification* couples opposite sides of the domain $\mathbb{P}$ and requires each of them to be closed. For instance, the west and east rims of $\mathbb{P}$ are identified if they define the same curve on S, i.e., $\forall v \in V, f_S(u_{\min}, v) = f_S(u_{\max}, v)$. Such a curve is called a *curve of identification*. In our running example of a parameterized sphere, we identify the west and the east rim of the parameter space, and the "international date line" is our curve of identification. A *torus* is modelled by identifying the two pairs of opposite rims. A *paraboloid* or *cone* can be modelled by identifying the vertical sides of $\mathbb{P}$ and contracting one of the horizontal sides to a point.



(a) (b)

Fig. 2. Possible DCEL representations of an arrangement of four lines in the plane

We simulate a sweep over $\tilde{S} = f_S(\mathbb{P} \setminus \partial\mathbb{P})$ in the interior of the parameter space. For this, we extend the definition of a *sweepable curve* to be a u-monotone curve whose interior is disjoint from the boundaries of $\mathbb{P}$. Isolated points and curves on $\partial\mathbb{P}$ can be viewed as any of its pre-images. In the preprocessing stage we split the input curves accordingly. In the example shown in Figure 1(b), both input curves c_1 and c_2 cross the line of identification, so each of them is split into two non-closed curved having its endpoints on the two copies of the line of identification. Note that any point lying in $\tilde{S}$ has a unique pre-image in $\mathbb{P}$, and a regular event is generated for it. For curves incident to the boundaries of $\mathbb{P}$ we generate events associated with $\langle\gamma, 0\rangle$ or with $\langle\gamma, 1\rangle$ to indicate γ's ends. As in the unbounded case, we are able to derive the correct order of events using the distinction between normal events that are associated with points on $\tilde{S}$ and events that occur on the boundaries of the parameter space. The same set of additional geometric predicates — namely, comparison of curve-ends on the boundary — is required; see [5] for more details. Similar to the unbounded case, we compare such curve ends in an ε-environment of the boundary; see the examples depicted in Figure 1(b) and (c).

The sweep proceeds exactly as the standard Bentley–Ottmann sweep does. Note however that if we have k sweepable curves incident to a point in $S \setminus \tilde{S}$ (namely a contraction point or a point on the curve of identification), we handle k separate events that relate to this point. In the next section we explain how we "tie all the loose ends" left out by the sweep procedure and construct a well-defined DCEL that represents an arrangement of curves on S.

4 Topology-Traits Classes

As mentioned above, we use a sweep-line visitor to process the topological information gathered in the course of the sweep (the "canonical ouput"), and construct a DCEL that represents the arrangement of the input curves. While in the case of an arrangement of *bounded* planar curves the DCEL structure is unique and well-defined (see, e.g., [16]), already for unbounded curves in the

plane we have a choice of representations, in particular for the representation of the unbounded faces. Figure 2(a) demonstrates one possibility, where we use an implicit bounding rectangle embedded in the DCEL structure using *fictitious* edges that are not associated with any concrete planar curve (this is the representation used in CGAL Version 3.3 [15]). An alternative representation is shown in Figure 2(b). It uses a single *vertex at infinity*, such that all edges that represent unbounded curves are incident to this vertex. Both alternatives are legitimate and each could be more suitable than the other in different situations.

Our implementation aims for flexibility and modularity. For example, we want to give the user the choice between different DCEL-representations as just discussed, and we want to encapsulate the geometric and topological information into a compact interface.

In addition to the *geometry-traits class*, which encapsulates the geometry of the curves that induce the arrangement (see Section 2), we introduce the concept of a *topology-traits class*, which encapsulates the topology of the surface on which the arrangement is embedded. The topology-traits class determines the underlying DCEL representation of the arrangement. In this traits class it is specified whether identifications or contractions take place and whether the parameter space is bounded or unbounded; see [5] for the exact details. Recall that there may be multiple events incident to the same point on the surface boundary. The sweep-line visitor uses the topology-traits class to determine the DCEL feature that corresponds to a curve-end incident to the boundary, so that all these curves will eventually coincide with a single vertex.

For example, if we sweep over the cylinder depicted in Figure 1(b), a vertex w is created on the curve of identification when we insert c_1^ℓ into the arrangement. The topology-traits class keeps track of this vertex, so it will associate w as the minimal end of c_2^ℓ and as the maximal end of c_1^r and c_2^r. Similarly, in the example shown in Figure 1(c), the north pole will eventually be represented as a single DCEL vertex, with c_1 and c_2 incident to it.

We have implemented four topology classes and the corresponding geometry classes: bounded curves in the plane, unbounded curves in the plane, arcs of great circles on a sphere, and intersection curves of quadrics on a quadric. The design of the specialized spherical-topology traits-class pretty much follows the examples we gave throughout the last two sections. We next discuss the topology-traits class for quadric surfaces in more depth.

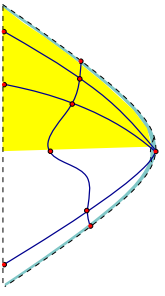
Example: Arrangements of Intersection Curves on a Quadric

A quadratic surface, *quadric* for short, is an algebraic surface defined by the zero set of a trivariate polynomial of degree 2: $\mathcal{Q}(x, y, z) \in \mathbb{Q}[x, y, z]$.We present an implementation of the two-dimensional arrangement embedded on a *base quadric* $\mathcal{Q}_0$, induced by intersections of quadrics $\mathcal{Q}_1, \ldots \mathcal{Q}_n$ with $\mathcal{Q}_0$. Our implementation handles all degeneracies, and is exact as long as the underlying operations are carried out using exact algebraic number types. We mention that some quadrics (e.g., hyperboloids of two sheets, hyperbolic cones) comprise two connected component. For simplicity, we consider each such component as a separate surface.

Non xy-functional quadratic surfaces can be subdivided into two xy-functional surfaces ($z = f(x, y)$) by a single continuous curve (e.g., the equator subdivides a sphere into two xy-monotone hemispheres), given by $\mathrm{sil}(\mathcal{Q}) = \gcd\left(\mathcal{Q}, \frac{\partial \mathcal{Q}}{\partial z}\right)$. The projection of such a curve onto the xy-plane is called a *silhouette curve*. The projection of an intersection curve between two quadrics onto the xy-plane is a planar algebraic curve of degree at most 4, which can be be subdivided into sweepable curves by intersecting it with $\mathrm{sil}(\mathcal{Q}_0)$. The interior of each such sweepable curve can then be uniquely assigned to the *lower part* or to the *upper part* of the base quadric $\mathcal{Q}_0$. Further details can be found in [6].

In our current implementation we only allow *ellipsoids*, *elliptic cylinders*, and *elliptic paraboloids* for the embedding surface $\mathcal{Q}_0$, as these types are nicely parameterizable by $\mathbb{P} = [l, r] \times [0, 2\pi)$, with $l, r \in \mathbb{R} \cup \{\pm\infty\}$, using $f_S(u, v) = (u, y(v), r(u, y(v), -\sin v))$. We define $y(v) = y_{\min} + (\sin \frac{v}{2})(y_{\max} - y_{\min})$, where $[y_{\min}, y_{\max}]$ denotes the y-range of the ellipse that $\mathcal{Q}_0$ induces on the plane $x = u$. The function $r(x, y, s)$ returns the minimal ($s \leq 0$) or maximal ($s > 0$) value of $\mathcal{R}_{\mathcal{Q},x,y} := \{z \,|\, \mathcal{Q}(x, y, z) = 0\}$, $|\mathcal{R}_{\mathcal{Q},x,y}| \leq 2$. For $\mathcal{Q}_0$, we detect open boundaries or contraction points in u and a curve of identification in v. Note that the quadrics $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ intersecting the base quadric $\mathcal{Q}_0$ can be arbitrary.

We partition the parameter space of v into two ranges $V_1 := [0, \pi]$ and $V_2 := (\pi, 2\pi)$. Given a point $p(u, v)$ on the base surface $\mathcal{Q}_0$, the *level* of p is $\ell \in \{1, 2\}$ if $v \in V_\ell$ (see the lightly shaded area in the figure to the right, which illustrates this partitioning for an arrangement on a paraboloid). Our geometry-traits class represents a sweepable intersection curve by its projected curve, the level of its projected curve (that needs to be unique in the interior), and its two endpoints. A point $p_i = (u_i, v_i)$ is represented by its projection (x_i, y_i) onto the xy-plane and its level ℓ_i. Given two points p_1 and p_2, their lexicographic order in parameter space is first given by the order of $x_1 = u_1$ and $x_2 = u_2$, and if $x_1 = x_2$ we infer the v-order from (y_1, ℓ_1), (y_2, ℓ_2): if $\ell_1 < \ell_2$ then $p_1 < p_2$, else if $\ell_1 = \ell_2$, then their v-order is identical to their y-order if $\ell_1 = 1$, and opposite to the their y-order if $\ell_1 = 2$.



The topology of an arrangement embedded on a base quadric $\mathcal{Q}_0$ requires special handling. The initialization of the DCEL consists of the creation of a single face, which can be *bounded* or *unbounded* depending on $\mathcal{Q}_0$. For the west and east rim of $\mathbb{P}$, two *boundary vertices*, named v_{left} and v_{right}, are used to record incidences to either points of contraction or to open boundaries. For an ellipsoid both represent points of contraction; for a paraboloid we have one vertex that corresponds to a contraction point and another that represents the unbounded side; for a cylinder, both vertices represent open (unbounded) boundaries. The south and north rims of the parameter space are identified. The topology-traits class maintains a sequence of vertices that lie on the curve of identification, sorted by their u-order. This sequence enables the easy identification of different events that correspond to the same point along this curve. We note that comparisons

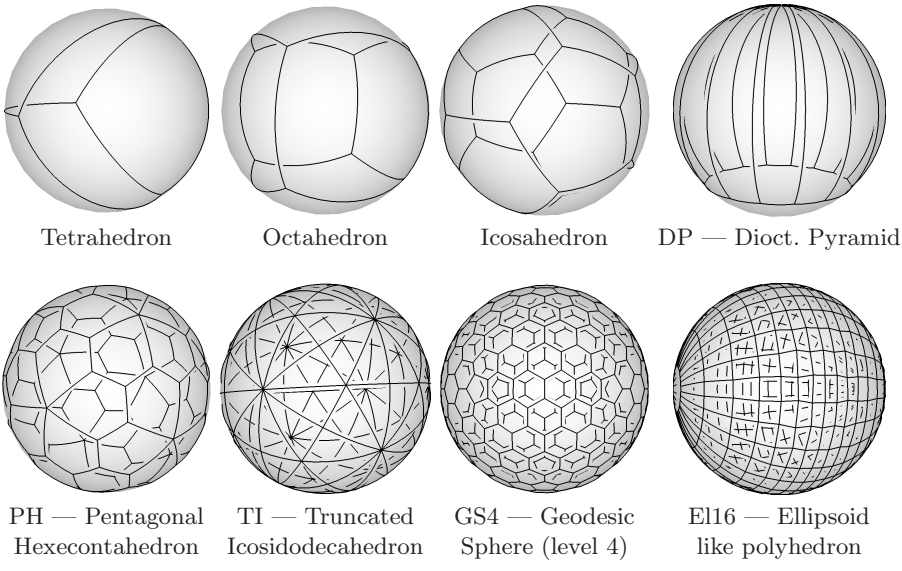


Fig. 3. Gaussian maps of various polyhedra

that require geometric knowledge are forwarded to the geometry-traits class, which in turn is implemented using the QuadriX library of Exacus [4].

5 Experimental Results

5.1 Arrangements of Great Arcs on Spheres

We demonstrate the usage of arrangements of arcs of great circles on the unit sphere through the construction of the *Gaussian map* [8] of convex polyhedra, *polytopes* for short. The geometry-traits class defines the point type to be a direction in $\mathbb{R}^3$, representing the place where the a vector emanating from the origin in the relevant direction pierces the sphere. An arc of a great circle is represented by its two endpoints, and by the plane that contains the endpoint directions and goes through the origin. The orientation of the plane determines which one of the two great arcs defined by the endpoints is considered. This representation enables an exact yet efficient implementation of all geometric operations required by the geometry-traits class using *exact rational arithmetic*, as normalizing directions and planes is completely avoided.

The overlay of the Gaussian maps of two polytopes P and Q identifies all the pairs of features of P and Q respectively that have common supporting planes, as they occupy the same space on the unit sphere, thus, identifying all the pairwise features that contribute to the boundary of the *Minkowski sum* of P and Q, namely $P \oplus Q = \{p + q \mid p \in P, q \in Q\}$.

We have created a large database of models of polytopes. The table below lists, for a small subset of our polytope collection, the number of features in the

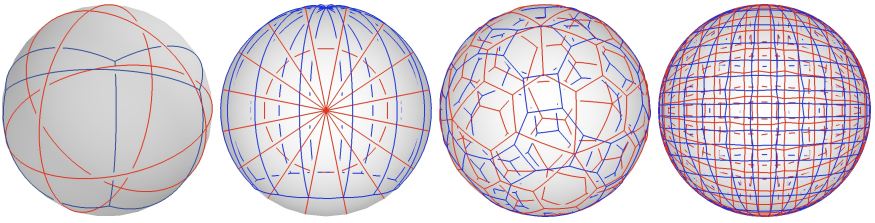


Fig. 4. Gaussian maps of the Minkowski sums

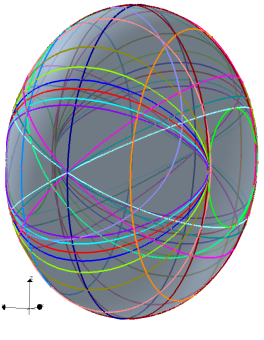
arrangement of arcs of great circles embedded on a sphere that represents the Gaussian map of each polytope. Recall that the number of vertices (**V**), edges (**E**), and faces (**F**) of the Gaussian map is equal to the number of facets, the number of edges, and the number of vertices in the primal representation, respectively. The table also lists the time in seconds (**t**) it takes to construct the arrangement once the intermediate polyhedron is in place, on a Pentium PC clocked at 1.7 GHz.

Object	V	E	F	t
Tetrahedron	4	6	4	0.01
Octahedron	6	12	6	0.01
Icosahedron	20	30	12	0.02
DP	17	32	17	0.06
PH	60	150	92	0.13
TI	62	180	120	0.33
GS4	500	750	252	0.56
El16	512	992	482	0.93

We have also conducted a few experiments that construct the Minkowski sums of pairs of various polyhedra in our collection. This demonstrates the successful overlay of pairs of arrangements on a sphere. The table on the right lists the number of features (**V**, **E**, **F**) in the arrangement that represents the Gaussian map of the Minkowski sum and the time in seconds (**t**) it takes to construct the arrangement once the Gaussian maps of the summands are in place. The prefix O before an object name indicates an orthogonal polyhedron. *These performance results are preliminary. We expect the time consumption to reduce significantly once all filtering steps are applied.*

Object 1	Object 2	V	E	F	t
Tetra.	Cube	14	28	16	0.09
DP	ODP	131	130	132	0.63
PH	TI	248	293	340	2.09
El16	OEl16	2260	2290	2322	17.07

5.2 Arrangements on Quadrics

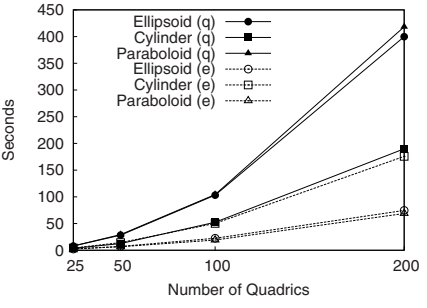


As we mentioned before, our implementation of the quadrics-traits classes is complete, and can handle all kind of degeneracies in a robust manner. The figure on the right shows the arrangement induced by 23 ellipsoids in degenerate position on a base ellipsoid. This highly degenerate arrangement is successfully constructed by our software.

We also measured the performance when computing the arrangement on given base quadrics induced by intersections with other quadrics. As base quadrics we created a random ellipsoid, a random cylinder, and a random paraboloid. These quadrics are intersected by

two different families of random quadrics. The first family consists of sets with up to 200 intersecting generic quadrics, sets of the other family include up to 200 ellipsoids intersecting each of the base quadrics. The coefficients of all quadrics are 10-bit integers. All performance checks are executed on a 3.0 GHz Pentium IV machine with 2 MB of cache, with the exact arithmetic number types provided by LEDA (Version 4.4.1).

Table 1 shows the number of arrangement features, as well as time consumption in seconds for selected instances. The figure on the right illustrates the average running time on up to 5 instances containing sets of ellipsoids (e) and general quadrics (q) of different sizes intersecting different base quadrics. Growth is super linear in the number of quadrics, as one expects.



Clearly, the more complex the arrangement, the more time is required to compute it. To give a better feeling for the relative time consumption, we indicate the time spent for each pair of half-edges in the DCEL of the computed arrangement. This time varies in the narrow range between 2.5 ms and 6.0 ms. Other parameters have significant effect on the running time as well, for example the bit-size of the coefficients of the intersection curves.

Table 1. Performance measures for arrangements induced on three base quadrics by intersections with 50 or 200 quadrics (q), or ellipsoids (e)

Base	Ellipsoid				Cylinder				Paraboloid			
Data	V	E	F	t	V	E	F	t	V	E	F	t
q50	5722	10442	4722	28.3	1714	3082	1370	12.5	5992	10934	4944	29.3
q200	79532	155176	75646	399.8	27849	54062	26214	189.9	82914	161788	78874	418.3
e50	870	1526	658	7.2	1812	3252	1442	14.4	666	1092	428	6.6
e200	10330	19742	9414	74.6	24528	47396	22870	175.8	9172	17358	8189	68.8

6 Conclusions and Future Work

We describe a general framework, together with a generic software package, for computing and maintaining 2D arrangements of arbitrary curves embedded on a class of parametric surfaces. We pay special attention to code reuse, which allows the development of traits classes for handling new families of curves and new surface topologies in a straightforward manner. Such developments benefit from a highly efficient code base for the main arrangement-related algorithms.

Single- vs. Multi-domain Surfaces: In this work we focus on the single domain case, namely our parameter space is represented by a single rectangle: $\mathbb{P} = U \times V$, as described in Section 3. Our major future goal is to extend the framework to handle general smooth surfaces, which can be conveniently represented by a collection of domains, each of which supported by a rectangular parameter space.

The individual parameter spaces are glued together according to the topology of the surface and therefore will naturally be described in, and handled by, an extension of the topology-traits concept introduced in this paper.

References

1. Agarwal, P.K., Sharir, M.: Arrangements and their applications. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 49–119. Elsevier, Amsterdam (2000)
2. Andrade, M.V.A., Stolfi, J.: Exact algorithms for circles on the sphere. Internat. J. Comput. Geom. Appl. 11(3), 267–290 (2001)
3. Bentley, J.L., Ottmann, T.: Algorithms for reporting and counting geometric intersections. IEEE Trans. Computers 28(9), 643–647 (1979)
4. Berberich, E., Eigenwillig, A., Hemmer, M., Hert, S., Kettner, L., Mehlhorn, K., Reichel, J., Schmitt, S., Schömer, E., Wolpert, N.: EXACUS: Efficient and exact algorithms for curves and surfaces. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 155–166. Springer, Heidelberg (2005)
5. Berberich, E., Fogel, E., Halperin, D., Mehlhorn, K., Wein, R.: A general framework for processing a set of curves defined on a continuous 2D parametric surface (2007), http://www.cs.tau.ac.il/cgal/Projects/arr_on_surf.php
6. Berberich, E., Hemmer, M., Kettner, L., Schömer, E., Wolpert, N.: An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In: Proc. 21st SCG, pp. 99–106 (2005)
7. Cazals, F., Loriot, S.: Computing the exact arrangement of circles on a sphere, with applications in structural biology. Technical Report 6049, Inria Sophia-Antipolis (2006)
8. Fogel, E., Halperin, D.: Exact and efficient construction of Minkowski sums of convex polyhedra with applications. In: Proc. 8th ALENEX (2006)
9. Fogel, E., Halperin, D., Kettner, L., Teillaud, M., Wein, R., Wolpert, N.: Arrangements. In: Boissonnat, J.-D., Teillaud, M. (eds.) Effective Computational Geometry for Curves and Surfaces, ch. 1, pp. 1–66. Springer, Heidelberg (2006)
10. Halperin, D.: Arrangements (chapter 24). In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, 2nd edn., ch. 24, pp. 529–562. Chapman & Hall/CRC (2004)
11. Halperin, D., Shelton, C.R.: A perturbation scheme for spherical arrangements with application to molecular modeling. Comput. Geom. Theory Appl. 10, 273–287 (1998)
12. Mehlhorn, K., Seel, M.: Infimaximal frames: A technique for making lines look like segments. J. Comput. Geom. Appl. 13(3), 241–255 (2003)
13. Meyerovitch, M.: Robust, generic and efficient construction of envelopes of surfaces in three-dimensional space. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 792–803. Springer, Heidelberg (2006)
14. Snoeyink, J., Hershberger, J.: Sweeping arrangements of curves. In: Proc. 5th SCG, pp. 354–363 (1989)
15. Wein, R., Fogel, E., Zukerman, B., Halperin, D.: 2D arrangements. In: Cgal-3.3 User and Reference Manual (2007), http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/Arrangement_2/Chapter_main.html
16. Wein, R., Fogel, E., Zukerman, B., Halperin, D.: Advanced programming techniques applied to CGAL's arrangement package. Comput. Geom. Theory Appl. 38(1–2), 37–63 (2007)

Fast and Compact Oracles for Approximate Distances in Planar Graphs

Laurent Flindt Muller[1] and Martin Zachariasen[2]

[1] Transvision, Vermundsgade 40D, DK-2100 Copenhagen, Denmark
laurent.flindt.muller@gmail.com
[2] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark
martinz@diku.dk

Abstract. We present an experimental evaluation of an approximate distance oracle recently suggested by Thorup [1] for undirected planar graphs. The oracle uses the existence of graph separators for planar graphs, discovered by Lipton and Tarjan [2], in order to divide the graph into smaller subgraphs. For a planar graph with n nodes, the algorithmic variant considered uses $O(n(\log n)^3/\epsilon)$ preprocessing time and $O(n(\log n)^2/\epsilon)$ space to answer factor $(1+\epsilon)$ distance queries in $O((\log n)^2/\epsilon)$ time. By performing experiments on randomly generated planar graphs and on planar graphs derived from real world road networks, we investigate some key characteristics of the oracle, such as preprocessing time, query time, precision, and characteristics related to the underlying data structure, including space consumption. For graphs with one million nodes, the average query time is less than $20\mu s$.

1 Introduction

Computation of shortest path distances in road networks has applications in traffic simulation and logistics optimization. Often a huge number of *distances* are needed, rather than actual shortest paths. A *distance oracle* is a data structure with an accompanying algorithm that can answer distance queries quickly (preferably in constant time) by preprocessing the graph and storing additional information [3]. For real world applications the preprocessing time and the space consumption both have to be near-linear in the size of the graph.

There has recently been a significant development in the practical construction of distance oracles that can answer (exact) distance queries quickly in arbitrary directed graphs. One approach is to identify so-called *landmarks* for which the distances to all other nodes are precomputed; these computed distances can then be used to speed up Dijkstra's algorithm using goal directed search [4]. Another technique uses the notion of *reach* [5], which is a measure of how far away the endpoints of any shortest path through a given node can be. Reach-based pruning is, like the use of landmarks, a highly effective method to speed-up Dijkstra's algorithm [6].

Sanders and Schultes [7,8] presented a technique that exploits the hierarchy inherent in road networks. In the preprocessing phase the method constructs

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 657–668, 2007.

a multilevel network in which exact shortest path queries subsequently can be answered quickly. Combined with the notion of *transit nodes* [9,10], distance queries in road networks with several million nodes can typically be answered in less than $10\mu s$.

Real world road networks are almost planar (with exceptions such as bridges and no-left-turns). Furthermore, the edge weights are essentially symmetric, so assuming that road networks are undirected planar graphs does not lead to particularly large errors — at least for long range distance queries. For undirected planar graphs, Fakcharoenphol and Rao [11] gave a distance oracle with $O(n \log^3 n)$ preprocessing time that answers (exact) distance queries in $O(\sqrt{n} \log^2 n)$ time, and it appears to be hard to improve this query time bound significantly using label-based methods [12].

If we settle for an *approximate distance oracle* that can answer distance queries within a factor $(1 + \epsilon)$ of the true distance (for any $\epsilon > 0$), much better results exist for undirected planar graphs. Klein [13] and Thorup [1] have independently suggested $O(n(\log n)/\epsilon)$-space approximate distance oracles answering factor $(1 + \epsilon)$ distance queries in $O(1/\epsilon)$ time. Both methods use the existence of graph separators for planar graphs, proved by Lipton and Tarjan [2], in order to divide the graph into smaller subgraphs. Then, for each node in the graph, the distances to a subset of the separator nodes are computed in a preprocessing step; these distances make it possible to answer approximate distance queries very quickly for any pair of nodes. The oracle given by Klein [13] has apparently been implemented as part of a commercial product, but to the best of our knowledge no publicly available experimental results are known.

The purpose of our paper is to perform an experimental study of a (simple) variant of the Thorup-oracle. In a sense we attempt to close part of the gap between theory and practice in distance oracles: The worst-case bounds for the highly effective practical algorithms [9,10] virtually give no insight into the practical behavior of the algorithms, and on the other hand, no experimental results are available for the theoretically superior algorithms of Thorup and Klein.

For a planar graph with n nodes, the algorithmic variant considered in our paper uses $O(n(\log n)^3/\epsilon)$ preprocessing time and $O(n(\log n)^2/\epsilon)$ space to answer factor $(1+\epsilon)$ distance queries in $O((\log n)^2/\epsilon)$ time. We have performed experiments on randomly generated planar graphs and on planar graphs derived from real world road networks. Key characteristics of the oracle, such as preprocessing time, query time, precision, and characteristics related to the underlying data structure, including space consumption, are studied. For graphs with one million nodes, the average query time is less than $20\mu s$. Although our query times are similar to the query times of the best practical algorithms, this comes at the cost of non-exact distances, and higher preprocessing and space consumption.

The paper is organized as follows. In Section 2 we present the approximate distance oracle of Thorup and discuss some important algorithmic and implementation details. In Section 3 we discuss our experimental setup, including the selection of test instances. Our experimental results are presented and discussed in Section 4, and concluding remarks are given in Section 5.

2 Approximate Distance Oracle

We now give an overview of the theoretical foundations of the oracle. We start by introducing the notion of a graph separator and describe how such a separator can be used to answer distance queries. Then we go on to describe ϵ-covers and finally describe the oracle itself. Basic graph terminology is used [2,14]; recall that a graph is called *planar* if its edges and nodes can be embedded in the plane such that no two edges cross or overlap.

2.1 Graph Separators

A set of nodes C in a graph G is called a *graph separator* if the removal of C from G results in G being split into components, the size of each being at most some constant fraction of the size of G (where the size refers to the number of nodes in the graph). To be of interest C must be "small".

Lipton and Tarjan [2] proved that it is possible to construct a graph separator for any planar graph in linear time. The construction essentially consists of finding a (rooted) shortest path tree of the graph and then selecting two root paths which separate the graph into components of size no greater than 2/3 the original size. Thorup [1] showed that this construction can be extended to selecting three root paths, where each component has size no greater than 1/2 the original size. We call such root paths *separator paths*.

Assume that we have two nodes u and v lying in two different components, resulting from the removal of a separator path Q. The observation is that any (shortest) path between u and v must cross a node in Q. Assume that we at some previous stage had computed distances from u and v to each node in Q. Now finding the distance between u and v is a simple task of running through the nodes of Q finding the minimum combined distance (see figure 1). This is in essence the idea behind the oracle. For each node u, we construct a set $S(u)$ consisting of separator paths, such that given any other node v the intersection $S(u) \cap S(v)$ consists of separator paths which, if removed from G, separate u from v. Thus a shortest path from u to v must cross a node in one of these paths. For each node u the distances to nodes in the separator paths in $S(u)$ are precomputed. Computing the distance between u and v then consists of finding the intersection $S(u) \cap S(v)$, then finding the minimum distance across each separator path in this intersection and finally returning the overall minimum distance.

2.2 ϵ-Covers

The most expensive part of the algorithm sketched above is precomputing distances from each node to nodes in the separator paths. In order to reduce this number of nodes, Thorup introduces the notion of ϵ-covers.

Let $G = (V, E)$ be a planar graph and let Q be a shortest path in G. A *connection* from $v \in V$ to Q is a pair in $\{v\} \times V(Q)$. The length $l(v, a)$ of a connection (v, a) is equal to the distance $\delta(v, a)$ between v and a in G. We

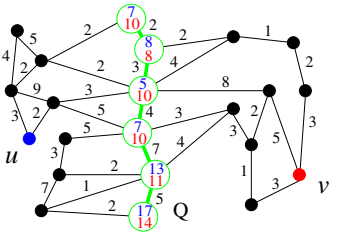


Fig. 1. Shows how a separator path can be used to compute the distance between two nodes u and v. The large nodes are the separator nodes and the topmost number is the distance from u, while the bottommost number is the distance from v. The min distance between u and v in the example is 15.

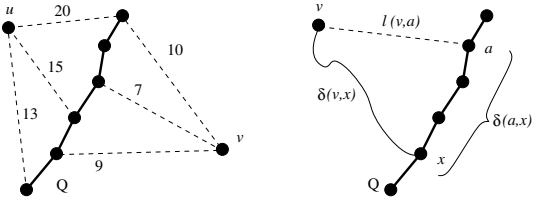


Fig. 2. Connections within the ϵ-covers $C(Q, u)$ and $C(Q, v)$ (left). Illustration of a connection (v, a) that ϵ-covers another connection (v, x) (right).

denote a set of connections from v to Q by $C(Q, v)$ (see figure 2, left). When computing distances across Q, only these connections are allowed to be used.

Let $\epsilon > 0$. We say that a set $C(Q, v)$ of connections from v to Q ϵ-covers Q, if there for any node $x \in V(Q)$ is a connection $(v, a) \in C(Q, v)$ such that (see figure 2, right): $l(v, a) + \delta(a, x) \leq \epsilon \cdot l(v, a) + \delta(v, x)$.

Thorup showed that for any node v and shortest path Q, there exists an ϵ-cover of size $O(\frac{1}{\epsilon})$ [1]. It is further shown that given two nodes u and v, and some separator path Q, the (minimum) distance between u and v across Q can be approximated within a factor $(1 + \epsilon)$ by using only the connections from $C(Q, u)$ and $C(Q, v)$.

2.3 Oracle

The preprocessing stage of the oracle proceeds as follows: Given a connected, undirected planar graph G, find up to three separator paths, Q_1, Q_2, Q_3, separating G into components of at most half the size. For each node v in G, construct the ϵ-cover $C(Q_i, v)$, $i = 1, 2, 3$. Proceed recursively on each of the components resulting from removing Q_1, Q_2, Q_3 from the graph. When the preprocessing terminates, we have for each node constructed a set of ϵ-covers, which can be used to answer distance queries. For each connection (u, a) in these ϵ-covers we store the distance from u to a and the distance from the first node in the separator path to a.

Consider two nodes u and v. At some point of the recursion these nodes get separated. When this happens, all ϵ-covers for u and v constructed so far have been coverings of the same set of separator paths: If $C(Q_{i_1}, u), \ldots, C(Q_{i_k}, u)$ and $C(Q_{j_1}, v), \ldots, C(Q_{j_l}, v)$ are the ϵ-covers constructed, then $k = l$ and $Q_{i_m} = Q_{j_m}$ for $m = 1, \ldots, k$. These are exactly the ϵ-covers which have to be checked in order to determine the distance between u and v. We call this number of ϵ-covers the *separation index* of u and v.

When the preprocessing is done, query-answering proceeds as follows: Given two nodes u and v, find the separation index k. For $i = 1, \ldots, k$ find the minimum distance across Q_i using the pair $C(Q_i, u), C(Q_i, v)$. Return the overall minimum distance found.

There are a number of implementation details which affect the overall time-complexity. When building ϵ-covers, any algorithm for finding shortest paths in (planar) graphs can be used. One should bear in mind that this algorithm will be called often and the hidden constants should be small. We employ Dijkstra's algorithm [14], since this is a simple and fast algorithm. On planar graphs it has a time-complexity of $O(n \log n)$, where n is the number of nodes, and this results in a time-complexity of $O(n(\log n)^3/\epsilon)$ for the preprocessing stage.

Another detail is how to find the separation index when answering distance queries; in order to find this index, each recursion is given a strictly increasing number. Each node maintains a list of recursion numbers representing the recursions in which it has participated. With each recursion number is stored the total number of separator paths generated in that recursion, plus all previous recursions in which the node has participated. In order to find the separation index of two nodes u and v one needs to find the last common recursion number. This marks the point where u and v got separated. We call this number the *separation number* (when we have found the separation number, we also have the separation index). Given two lists of recursion numbers we use binary search to find the separation number. Since there are at most $O(\log n)$ separation numbers for each node, this search has time-complexity $O(\log \log n)$.

A final detail is the size of the ϵ-covers. The construction to produce ϵ-covers of constant size is a two step process. First covers of size $O(\log |V(Q)|/\epsilon)$ are constructed, where Q is a separator path. Then the size of these covers is reduced to $O(1/\epsilon)$. This last step is not performed in our implementation. The reason is, as we shall see later, that the covers produced by the first step alone are already small (around 11 connections in each cover for $\epsilon = 0.01$ and one million nodes). Moreover, in order to achieve ϵ-covers of size $O(1/\epsilon)$, one needs to run the oracle with $\epsilon' = \epsilon/2$, which increases the running time and the size of the ϵ-covers produced in the first step (and hence also the second step). We expect that for the sizes of graphs considered, the constant in front of $1/\epsilon$ does not differ enough from the value $\log |V(Q)|$ to justify the extra running time. This results in a $O((\log n)^2/\epsilon)$ time-complexity for answering queries ($O(\log n)$ ϵ-covers must be checked and each ϵ-cover contains $O(\log n/\epsilon)$ connections).

3 Experimental Setup

To examine the performance of the oracle, we have looked at two classes of graphs: (i) Graphs derived from real world road networks of the United States (made publicly available for the 9th Dimacs Implementation Challenge [15]). The graphs considered are planar and connected, and have the sizes: $321,270$ (BAY), $435,666$ (COL) and 1,070,376 (FLA). (We were unable to perform experiments on larger graphs due to memory constraints.) In the following, we call this class of graphs *real-world* instances. (ii) Randomly generated connected planar graphs generated using the `random_planar_graph()` function in LEDA [16]. Edge lengths corresponds to Euclidean distances, with a random variance of up to 10%. We expect this choice of edge lengths more closely reflects real world road graphs, as compared to random or exact Euclidean lengths. In the following, we call this class of graphs *near-Euclidean* instances.

When presenting query-times and precision, results have been divided into four groups. Each group corresponds to queries where the distance falls within a certain percentage range of the graph diameter, corresponding to 0%-25%, 25%-50%, 50%-75% and 75%-100% of the diameter. The size of each group of queries is capped at 1000 and filled up as follows: First the oracle is run with $\epsilon = 0.05$. Then random queries are generated and answered using this oracle. If the distance returned falls within the range of an unfilled group, then the exact distance is found using Dijkstra's algorithm. The process stops when all groups are filled or when 200.000 queries have been made.

The tests have been run on a machine running *Gentoo Base System version 1.12.6*. The compiler used is *GCC 3.4.6* with optimization flag `-O`. The machine has 2 GB of main memory and two *Intel Pentium 4 CPU 3GHz* processors.

4 Results and Discussion

In the following we present our results for the two classes of graphs. For each instance the preprocessing stage has been run a number of times (around 5) and the average taken. For each run ϵ has been set to 0.10, 0.05 and 0.01. We also examine the effect of setting $\epsilon = 0$, but due to memory constraints results are in this case only presented for the near-Euclidean graphs.

In order to keep memory-consumption low, ϵ-covers are regularly flushed to disk and in order to keep disc-usage low, the outputted files are compressed. The measured time includes these overheads.

In order to examine the query stage, the range groups described above are used. Again, the queries have been run repeatedly and the average taken.

4.1 Real-World Graphs

Figure 3 (left) shows the total CPU-time spent in the preprocessing stage when ϵ is varied. For the FLA instance, the preprocessing time varies from around $4000s$ for $\epsilon = 0.10$ to around $9000s$ for $\epsilon = 0.01$. The bulk of this time (around 90%) is spent constructing ϵ-covers.

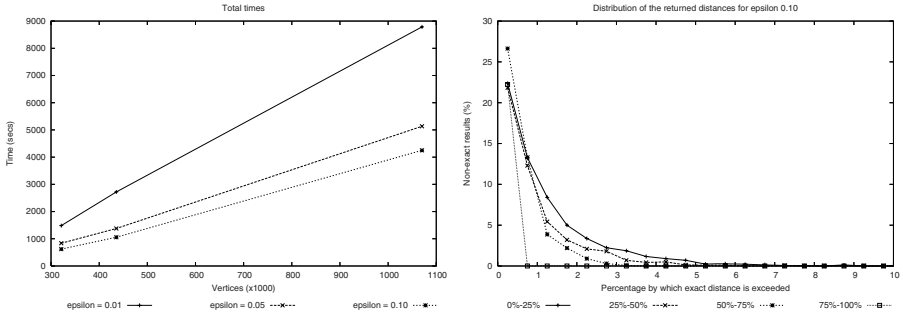


Fig. 3. Total time spent in the preprocessing stage (left) and distribution of distances returned for the *FLA* instance ($\epsilon = 0.10$), for the four ranges (right) for real-world graphs. (Note that there are only three data points for each ϵ-value, and that the points are connected only in order to improve readability).

Figure 4 shows the query-times for the first and the last range group (the results for the remaining two look very similar). The value of ϵ has a considerable effect. Query-times are below $7\mu s$ for $\epsilon = 0.10$, and increase to $20\mu s$ for $\epsilon = 0.01$. Generally finding shorter distances takes longer time. One explanation for this is that nodes that lie far apart get separated earlier, and thus fewer ϵ-covers need to be checked in order to answer a query.

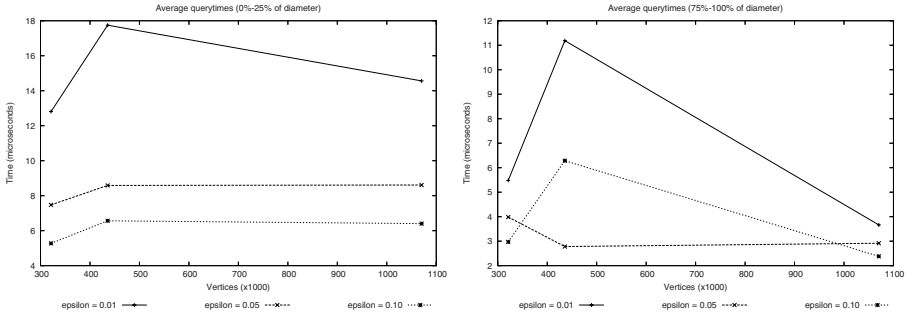


Fig. 4. Average query-times for distances lying in the interval 0%-25% (left) and 75%-100% (right) of the diameter (real-world)

We now investigate the precision of the oracle, by examining the results for the *FLA* instance with $\epsilon = 0.10$. Figure 3 (right) shows the results for the four range groups. The numbers indicate how many percent of the 1000 distances were approximated within $1\%, 2\%, 3\%$ etc, of the true distance. Only non-exact distance are displayed. As can be seen, the further the nodes lie from each other, the more accurate the distances returned are. Those distances that are not exact tend to be very close to the exact distance. The average error is less than 1%, so even with $\epsilon = 0.10$ one obtains very good approximations.

We now look at some of the characteristics of the oracle. Figure 5 (left) shows the average number of connections in each ϵ-cover when ϵ is varied. These ϵ-covers are on average surprisingly small. For the *FLA* instance with $\epsilon = 0.01$, there are 11 connections per ϵ-cover and for $\epsilon = 0.10$, there are 5. We remind the reader that the bound on the size of these covers is $O(\log V(Q)/\epsilon)$, where Q is the separator path. As mentioned earlier, since the ϵ-covers are already so small, we do not expect that using an algorithm with a better bound on the size will have any dramatic effect on the query-time.

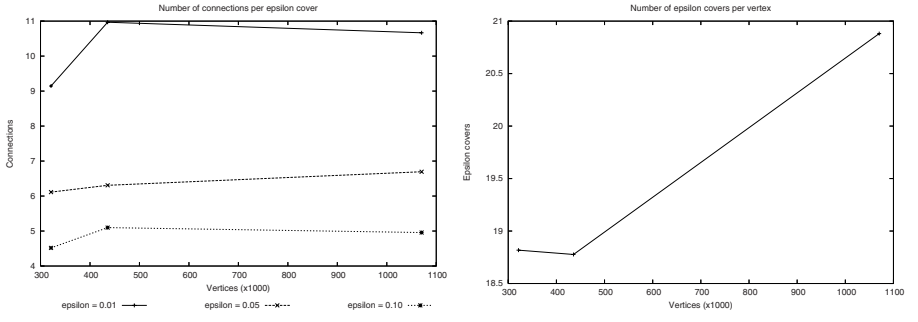


Fig. 5. Average number of ϵ-covers for each node (left) and average number of connections in each ϵ-cover (right) for real-world graphs

Figure 5 (right) shows the average number of ϵ-covers for each node (this number is independent of ϵ). This number lies between 18 and 21, so on average around 200 connections have to be examined in order to answer a query for $\epsilon = 0.01$.

The total number of connections constructed during preprocessing gives an indication of the memory consumption of the oracle. In the current implementation, one connection consists of two floats, which is 8 bytes in total. The total number of connections for the *FLA* instance with $\epsilon = 0.01$ is around 250 million, which results in a memory consumption of about $2GB$. The number of connections is strongly affected by ϵ. When ϵ is increased to 0.10 for the same instance, the number of connections drops to just under 100 million.

4.2 Near-Euclidean Graphs

In this section we look at the class of near-Euclidean graphs. There are some notable differences compared to the previous section. Figure 6 (left) shows the CPU-time spent in the preprocessing stage when ϵ is varied. The preprocessing time is within the same order of magnitude as for the real-world instances, but varying ϵ has less effect.

In order to examine the query-times, we proceed as for the real world graphs. In figure 7 the results for the first and the last range group are shown (the remaining two look very similar) On this class of graphs query-times are lower

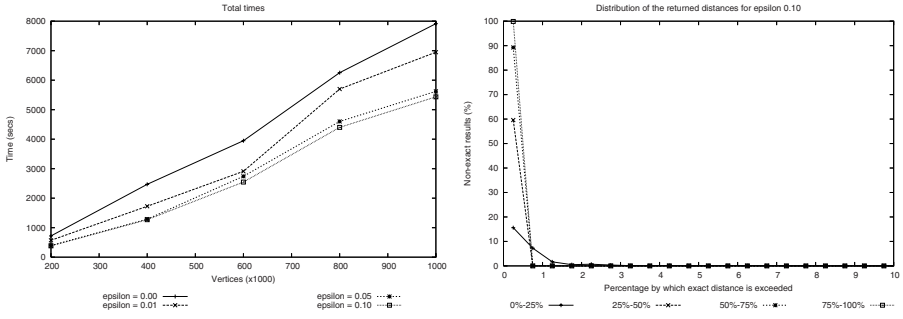


Fig. 6. Total time spent in the preprocessing stage (left) and distribution of distances returned for the instance with 1 million nodes ($\epsilon = 0.10$), for the four ranges (right) (near-Euclidean graphs)

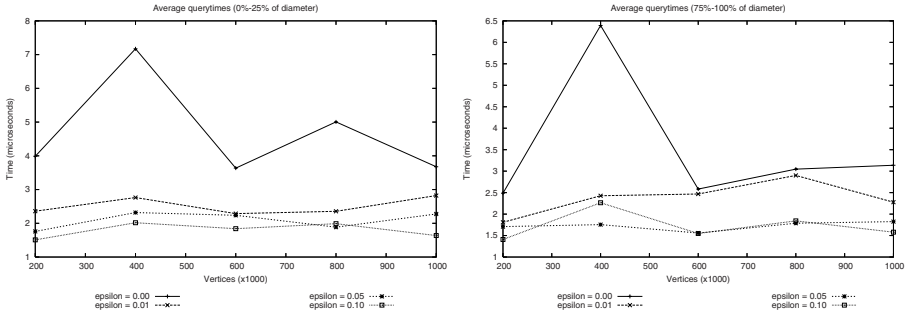


Fig. 7. The average query-times for distances lying in the interval 0%-25% (left) and 75%-100% (right) of the diameter (near-Euclidean graphs)

and range from around $2\mu s$ for $\epsilon = 0.10$ to around $3\mu s$ for $\epsilon = 0.01$ and increase to about $7\mu s$ for $\epsilon = 0$. The reason for the lower query-times will become clear later. On these instances, there is no noticeable difference w.r.t. range groups.

As for the real-world instances, we examine the precision of the oracle. Figure 6 (right) shows the results for the largest instance and for $\epsilon = 0.10$. The distances are even closer to the exact distance than for the real-world instances. In fact all distances in the three last groups lies within 0.5% of the exact distance.

Figure 8 (left) shows the average number of connections in each ϵ-cover when ϵ is varied. These are only about half the size of the covers produced for the real-world instances. Varying ϵ only has a minor effect on the size, which for the largest instance ranges from 3 for $\epsilon = 0.10$ to 6.5 for $\epsilon = 0$.

Figure 8 (right) shows the average number of ϵ-covers for each node. This number (between 17 and 23) is about the same as for the previous class. Together with the smaller ϵ-covers, this explains why we on this class see faster query-times than before. We believe that the ϵ-covers are smaller because the edge lengths are closer to fulfilling the triangle-inequality, and therefore it is easier for a connection to cover another.

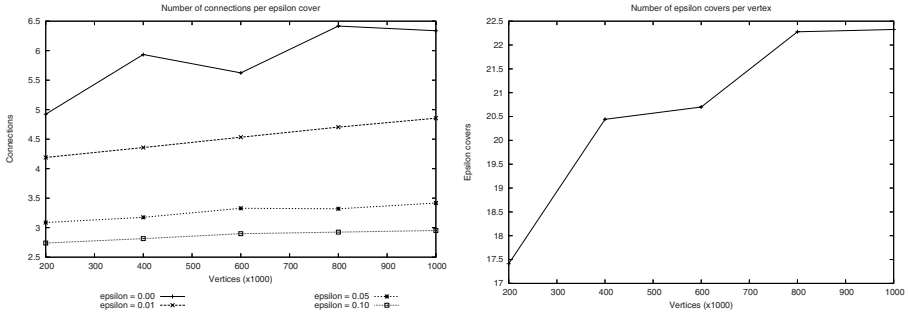


Fig. 8. Average number of connections in each ϵ-cover (left). Average number of ϵ-covers for each node for near-Euclidean graphs (right).

The total number of connections constructed is about half as large as for the real-world instances. This is as expected, since the ϵ-covers are about half the size. The number is still affected by ϵ and for the largest instance it ranges from around 60 million connections for $\epsilon = 0.10$ to around 110 million connections for $\epsilon = 0.01$ and 140 million for $\epsilon = 0$.

4.3 Comparison to an Alternative Oracle

We make a brief comparison to an alternative algorithmic variation of the Thorup-oracle. In the variant examined this far, the number of separator paths to check when answering a query is $O(\log n)$. Thorup [1] describes another variant, where this number is $O(1)$.

In this variant, alternating between two types of recursions, one can ensure that at most 12 separator paths needs to checked when answering a query, i.e., $|S(u) \cap S(v)| \leq 12$ for all u, v.

There are, however, certain drawbacks. The alternation between two types of recursions results in a doubling of the total number of connections having to be made, thus one should also expect a doubling of the space consumption and an increase of the total running time. Considering that memory consumption already is a major issue, this doubling is unfortunate.

Despite this we have made an implementation of this variant, hoping that it would result in a significant improvement of the query-time. Regrettably, this is not the case, as the query-times observed are within the same order of magnitude as the previous variant, while the preprocessing-time and memory consumption grew as expected. For the *FLA* instance ($\epsilon = 0.05$) the preprocessing time grew from $5000s$ to $16000s$, while the total number of connections grew from 150 million to 275 million.

A closer inspection of the number of ϵ-covers for each node gives a good insight as to the reason for not seeing any improvement in query-times. Looking at figure 5 (left) the average number of ϵ-covers for each node lies around 21 for the *FLA* instance. Thus on average 21 ϵ-covers need to be checked. It is this

number which is reduced to at most 12. The constructed ϵ-covers are of the same sizes, so one should not expect any significant gain for the query-times.

For the real-world instances considered (and near-Euclidean) the number of ϵ-covers for each node rises rather slowly with graph size (from around 18 for BAY to around 21 for FLA). So the reduction to size 12 would only begin to be noticeable for very large graphs.

For the graph sizes considered, we do not find that the rather limited improvement in query-time warrants the extra running time and space consumption.

5 Conclusion

In this paper we presented an experimental evaluation of an oracle recently suggested by Thorup [1]. For the examined class of real-world road network graphs, the observed query-times were less than $20\mu s$ ($\epsilon = 0.01$). The preprocessing time associated with the largest instance (one million nodes) was around $9000s$ and the memory needed to hold the generated connections was about $2GB$ for $\epsilon = 0.01$.

For the examined class of randomly generated near-Euclidean graphs, even faster query-times below $3\mu s$ were observed (still with $\epsilon = 0.01$). For the largest instance (with one million nodes) the preprocessing time was faster, around $7000s$, and memory consumption was about $1GB$ for $\epsilon = 0.01$. For this class of graphs, we also examined the effect of setting $\epsilon = 0$ (exact distance). This did not worsen the preprocessing time (around $8000s$), query-time (below $6.5\mu s$) and memory consumption (around $1.2GB$ for all connections) as much as could have been feared, which is interesting.

When examining the precision of the returned distances, we observed that these tend to lie very close to the exact distances, even for $\epsilon = 0.10$. This suggests that in practice one could set ϵ to a relatively large number and still get good approximations.

The strongest merit of this oracle is its fast query-times and guaranteed bounds. This though, comes at the expense of approximate (but near-optimal) distances, and a preprocessing time and memory consumption, which is rather large. Other recent approaches, such as *transit node* based algorithms [9,10] achieve similar results on road networks, but without sacrificing exactness. When compared to these approaches, the results for the oracle considered do not appear to be competitive.

Whether the preprocessing time and memory consumption may be improved is not yet clear. One way to reduce the former would be to experiment with other shortest path algorithms when constructing ϵ-covers. An improvement here would lead to a reduction in preprocessing time. In order to reduce the latter, one has to reduce the total number of connections needed (or reduce the space needed for each connection). Both Thorup [1] and Klein [13] show that it is possible to construct ϵ-covers of constant size. Further examination of the effect of this on the total number of connections is needed, but as argued in section 2.3 we do not expect the effect to be dramatic.

References

1. Thorup, M.: Compact Oracles for Reachability and Approximate Distances in Planar Digraphs. Journal of the ACM 51, 993–1024 (2004)
2. Lipton, R.J., Tarjan, R.E.: A Separator Theorem for Planar Graphs. SIAM Journal on Applied Mathematics 36, 177–189 (1979)
3. Thorup, M., Zwick, U.: Approximate Distance Oracles. Journal of the ACM 52, 1–24 (2005)
4. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A^* Meets Graph Theory. In: Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms, pp. 156–165. ACM Press, New York (2005)
5. Gutman, R.: Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In: Proceedings 6th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 100–111 (2004)
6. Goldberg, A.V., Kaplan, H., Werneck, R.: Reach for A^*: Efficient Point-to-Point Shortest Path Algorithms. In: Workshop on Algorithm Engineering and Experiments (2006)
7. Sanders, P., Schultes, D.: Highway Hierachies Hasten Exact Shortest Path Queries. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 568–579. Springer, Heidelberg (2005)
8. Sanders, P., Schultes, D.: Engineering Highway Hierachies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)
9. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast Routing in Road Networks with Transit Nodes. Science 316, 566 (2007)
10. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Time Shortest-Path Queries in Road Networks. In: Proc. 9th Workshop on Algorithm Engineering and Experimentation (ALENEX) (2007)
11. Fakcharoenphol, J., Rao, S.: Planar Graphs, Negative Weight Edges, Shortest Paths, and Near Linear Time. In: Proceedings of 42nd IEEE Symposium on Foundations of Computer Science, pp. 232–241. IEEE Computer Society Press, Los Alamitos (2001)
12. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance Labeling in Graphs. Journal of Algorithms 53, 85–112 (2004)
13. Klein, P.: Preprocessing an Undirected Planar Network to Enable Fast Approximate Distance Queries. In: Proceedings of the thirteenth annual ACM-SIAM Symposium on Discrete Algorithms, pp. 820–827. ACM Press, New York (2002)
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
15. Demetrescu, C., Goldberg, A.V., Johnson, D.: 9th DIMACS Implementation Challenge (2006), http://www.dis.uniroma1.it/~challenge9/
16. GMBH, A.S.S.: LEDA 5.2 (2007), http://www.algorithmic-solutions.com/

Exact Minkowksi Sums of Polyhedra and Exact and Efficient Decomposition of Polyhedra in Convex Pieces

Peter Hachenberger*

Department of Computing Science,
TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
phachenb@win.tue.nl

Abstract. We present the first exact and robust implementation of the 3D Minkowski sum of two non-convex polyhedra. Our implementation decomposes the two polyhedra into convex pieces, performs pairwise Minkowski sums on the convex pieces, and constructs their union. We achieve exactness and the handling of all degeneracies by building upon 3D Nef polyhedra as provided by CGAL. The implementation also supports open and closed polyhedra. This allows the handling of degenerate scenarios like the tight passage problem in robot motion planning.

The bottleneck of our approach is the union step. We address efficiency by optimizing this step by two means: we implement an efficient decomposition that yields a small amount of convex pieces, and develop, test and optimize multiple strategies for uniting the partial sums by consecutive binary union operations.

The decomposition that we implemented as part of the Minkowski sum is interesting in its own right. It is the first robust implementation of a decomposition of polyhedra into convex pieces that yields at most $O(r^2)$ pieces, where r is the number of edges whose adjacent facets comprise an angle of more than 180 degrees with respect to the interior of the polyhedron.

1 Introduction

The Minkowski sum of two point sets P and Q in $\mathbb{R}^d$, denoted by $P \oplus Q$, is defined as the set $\{p + q : p \in S_1, q \in S_2\}$. Minkowski sums are used in a wide range of applications such as robot motion planning [15], computer-aided design and manufacturing [8], penetration depth computation [14], offset computation [17], morphing [13], and mathematical morphological operations [18].

In several applications (e.g. in GIS or imaging) one deals with Minkowski sums of two-dimensional objects, and several implementations exist. When the two objects are non-convex polygons, two approaches are commonly used. The first approach computes the convolution of the boundary of two polygons [10].

* This work was partially supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 639.023.301.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 669–680, 2007.

The other approach decomposes both polygons into convex pieces, computes the pairwise Minkowski sums of the pieces, and unites the pairwise sums. Both approaches have also been studied and implemented in combination with exact geometric computation [1,20]. The *Library for Efficient Data Types and Algorithms* (LEDA)[1] offers an exact implementation based upon the first method. The *Computational Geometry Algorithm Library* (CGAL)[2] offers exact implementations of both methods.

There are also many applications, however, that require the computation of the Minkowski sums of three-dimensional objects. Examples can be found in CAD/CAM, assembly planning, and motion planning. Implementations of the 3D Minkowski sum exist, but they are neither exact nor robust. The most efficient such implementation is probably by Varadhan and Manocha [19]. It is based upon the convex decomposition approach as described above for the two-dimensional case. They guarantee the correct topology of the result, but are limited to manifold boundaries. Although many input objects are commonly two-manifolds, this limitation seems to be a major robustness issue, because the primitives of the union step, i.e., the Minkowski sum of two convex pieces, are not allowed to touch tangentially. Thus, at the moment there is no implementation available that is robust (that is, can deal with all possible degenerate cases), nor is there an implementation that is exact. This is the goal of our work: to provide a solution for the computation of Minkowski sums of 3D polyhedra that can handle all degenerate cases and is exact.

We present the first exact implementation of the Minkowski sum of two nonconvex polyhedra. Our implementation is based on the convex decomposition approach. For the union step we use 3D Nef polyhedra [11] as provided by CGAL, which provide exact and efficient Boolean operations and handle all degeneracies. Our solution handles regularized solids with open or closed boundary. As a consequence, it can also be applied to degenerate scenarios like the tight passage problem in robot motion planning.

In addition to exactness, we also emphasize efficiency. We mostly concentrate on optimizing the time needed by the union of the pairwise Minkowski sums of convex pieces, which is the bottleneck of the used approach. For reducing the runtime of the union it is essential to have a decomposition that yields a low number of convex pieces. The decomposition into a minimum number of convex pieces is known to be NP-hard [16]. More than 20 years ago Chazelle proposed a decomposition method, which generates $O(r^2)$ convex pieces in $O(nr^3)$ time and $O(nr^2)$ space, where n is the complexity of the polyhedron and r is the number of *reflex edges*—edges, whose adjacent facets form an angle larger than 180 degrees with respect to the interior of the polyhedron. However, no robust implementation of this algorithm is known. Most of the practical methods perform surface decomposition or tetrahedral volumetric decomposition [5,7,12]. These methods generate $O(n)$ convex pieces of constant complexity.

[1] <http://www.algorithmic-solutions.com>
[2] <http://www.cgal.org>

As part of our Minkowski sum, we present the first robust implementation of a decomposition of a non-convex 3D polyhedron into $O(r^2)$ convex pieces. We use Chazelle's main idea of inserting facets that resolve reflex edges. On the other hand, we construct the facets by a completely different method. Apart from the technical differences, we keep the actual amount of convex pieces low by scheduling the construction of the facets in an opportune way.

To optimize the union step itself, we develop different union strategies. The union of multiple polyhedra is done by consecutive binary union operations. Here, the order of these unions is essential for the runtime. Our union strategies use different heuristics to minimize the complexity of the intermediate results and to reduce the total amount of memory usage. We compare the efficiency of the heuristics experimentally.

The paper is organized as follows. In Section 2 we discuss how to efficiently decompose a non-convex polyhedron into convex pieces. The Minkowski sum of convex 3D polyhedra is discussed in Section 3. Section 4 compares multiple union strategies and refines the most promising. Also, we perform one larger experiments to get an idea of the performance. In Section 5, we briefly discuss the handling of tight passage problems. Finally, a conclusion is given in Section 6.

2 Decomposing a Polyhedron into Convex Pieces

The problem of partitioning a polyhedron into convex pieces is more complex than its two-dimensional counterpart. In general it is not possible to decompose a polyhedron into simplices, i.e., into tetrahedra, without introducing Steiner points [16]. The decomposition of a polyhedron into a minimum number of convex pieces is known to be NP-hard [16].

A basic decomposition method was introduced and analyzed by Chazelle [4]. The idea is to remove each *reflex edge*, i.e., each edge whose adjacent facets have an angle larger than 180 degree with respect to the interior of the polyhedron, by inserting an additional facet that cuts the angle into two parts smaller than 180 degrees. Chazelle showed that a polyhedron with input complexity n and r reflex edges, can be decomposed into $O(r^2)$ convex pieces in $O(nr^3)$ time and $O(nr^2)$ space. He also provided an example for which the bound of $O(r^2)$ convex sub-polyhedra is tight.

We follow the common decomposition approach of inserting only vertical facets usually denoted as walls. A *wall* $W(e)$ of some non-vertical edge e is a connected subset of the vertical plane p_e that supports e. Walls were first defined by Aronov and Sharir [2]. Because their definition was given for a decomposition of the three-dimensional space with respect to a set of triangles we adapt their definition to our problem as follows: Let $A(p_e)$ be the planar arrangement of the intersection of the polyhedron (including previously erected walls) with the vertical plane p_e through e. Then, the wall of $W(e)$ consists of all faces of $A(p_e)$ that are incident to e and inside the polyhedron. The left graphic of Figure 1 illustrates the planar arrangement $A(p_e)$ and the wall $W(e)$.

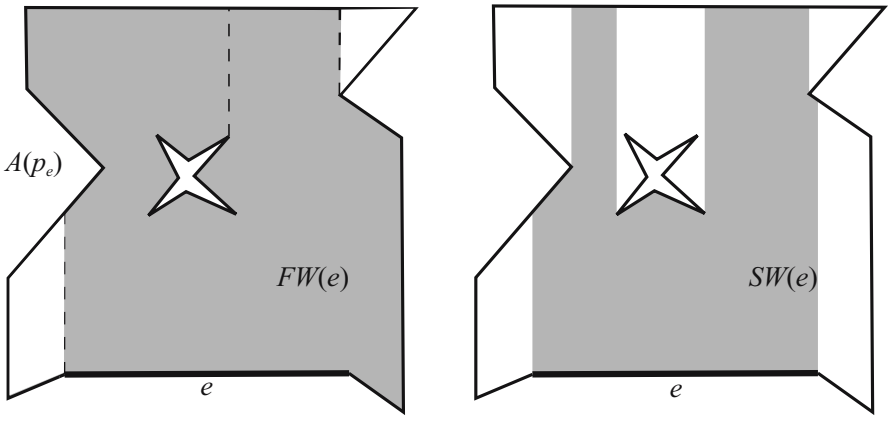


Fig. 1. *Left*: The flood wall $FW(e)$ consists of all facets adjacent to e in the planar arrangement $A(p_e)$ of the intersection of the polyhedron and the vertical plane p_e through e. The arrangement $A(p_e)$ may also contain previously inserted walls (dashed lines). *Right*: A sight wall $SW(e)$ covers all points that can be connected to e by a vertical edge without intersections.

Later de Berg, Guibas, and Halperin defined a vertical wall as the set of all points that can be connected to e via a vertical segment that does not intersect a face, edge, or vertex [6]. Adapting their definition to our setting, we consider points that can be connected to e via a vertical segment that completely lies within the polyhedron. The right graphic of Figure 1 illustrates the definition. To distinguish the two wall types, we further-on refer to the walls as defined in [2] as flood walls and to the walls as defined in [6] as sight walls.

The convex decomposition by vertical walls, also denoted as *vertical decomposition*, works in two steps. In the first step, vertical walls are erected for all non-vertical reflex edges. As a consequence, the decomposed volume becomes subdivided into cylindrical cells. In the second step, walls parallel to the yz-plane resolve the vertical reflex edges. Figure 2 illustrates the two steps of the vertical decomposition. Our decomposition deviates from the common approach to reduce the number of sub-polyhedra.

Using sight walls in the first phase often yields fewer, and never yields more pieces than using flood walls. This becomes clear if we consider the construction of two flood walls of reflex edges e and e'. Let us assume that $FW(e)$, if built first, seals a pocket in the boundary of the polyhedron. But if another flood wall $FW(e')$ is built before, it may intersect e and therefore split the pocket into two halves. Both halves of the pocket will later be sealed separately by the walls constructed for the two halves of e. Thus, depending on the building order of the flood walls, such a pocket can be decomposed into multiple pieces, although fewer pieces (or even one piece) are sufficient. Using sight walls instead, no $SW(e')$ splits the pocket into two halves; vertical segments cannot intersect such a pocket.

Because the reflex edges in the second phase are vertical, the definition of sight walls cannot be applied to these edges. Also, there is no unique vertical

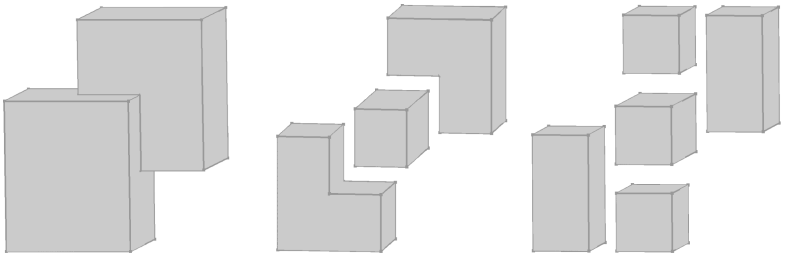


Fig. 2. Vertical decomposition based on the insertion of walls (viewed from the top). In the first step, the polyhedron is decomposed into xy-monotone sub-cells. Then, further vertical walls are inserted to subdivide the cells into convex sub-cells.

supporting plane that defines the flood wall of a reflex edge. The procedure of vertical decomposition suggests to create y-vertical flood walls. We deviate from this procedure to reduce the number of walls and therefore the number of sub-polyhedra. An y-vertical flood wall divides a volume into two or three parts. Instead, we insert a flood wall along one of the two facets adjacent to the reflex edge. This way, resolving a y-vertical reflex edge splits exactly one volume into two parts.

In the second step of the decomposition, walls can be built easily. Given a reflex edge e, let p_e be the plane in which we intend to build the wall $W(e)$, and let c be the cylindrical cell that will be decomposed by it. Then, $W(e)$ can be created by walking along the intersection of c with p_e and adding the incidences of the new facets to each encountered item. In the first step, the boundary of a wall is more complex. The boundary of $W(e)$ may not only consist of e and intersections with c, but also of intersections with other walls. If the walls are built in random order it is not guaranteed that those intersections exist when constructing a wall. Therefore, we cannot just walk along the boundary. To allow using the walk, we schedule the construction of the walls in such a way that all boundary parts of a wall exist in the moment of its construction. To do this, we must resolve mutually and cyclic dependencies.

The wall $W(e)$ of a reflex edge e may consist of two parts—of points below and points above e. We refer to these two parts as the lower wall $W^-(e)$ and upper wall $W^+(e)$. Often, the lower wall $W^-(e)$ is part of the boundary of some upper wall $W^+(e')$, or vice versa. Therefore, the lower and the upper walls are created in two separate sessions.

Starting with the lower parts, we want to sort the reflex edges, and thereby schedule their construction, from bottom to top. In general, the edges of a polyhedron—and the same holds for the reflex edges of a polyhedron—can not be sorted along a given direction. There can always be cyclic dependencies as illustrated by Figure 3. But, as we will see in the following, it is possible to resolve the dependencies.

We sort the reflex edges by their lower endpoints. As a result, every pair of reflex edges has one of three relations. If (a) they do not overlap vertically, the sorting schedules the edges well. The same holds, if they overlap vertically, but their

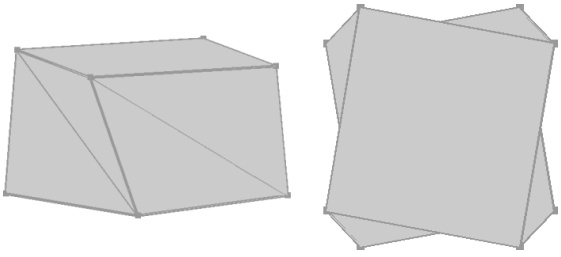


Fig. 3. Variation of Schönhardt polyhedron with a quadratic base viewed from the side and from the top. The diagonals of the sides are reflex edges, which are circular dependent on one another.

projection onto the xy-plane does not intersect. If (b) they overlap vertically and their projections onto the xy-plane intersect internally, the lower wall of one of them may be part of the other's boundary. Let's assume such an edge pair e and e', where $W^-(e)$ is part of the boundary of $W^-(e')$. If e has the smaller lower endpoint the schedule works nicely; otherwise the walk cannot proceed along $W^-(e')$'s boundary because $W^-(e)$ is missing. If (c) they share a common endpoint, the lower walls share a boundary edge. Their construction is mutually dependent.

To solve the problems described above, we cut some reflex edges into two or more parts and insert vertical edges from the endpoints of all reflex edges to the bottom of the polyhedron. The inserted vertical edges exactly resemble the common boundary parts of walls with common endpoints and therefore resolve the problem described in (c). Before inserting the vertical edges, we search the sorted list of reflex edges for situation (b). If we find an edge e part of the boundary of $W(e')$, but e' is scheduled before e, then we cut e at the intersection point v_i with the vertical plane $p_{e'}$ supporting e' and put the two edge halves at their proper positions into the sorted set. Later a vertical edge will also be inserted between v_i and the bottom of the polyhedron. It exactly resembles the intersection between $W^-(e)$ and $W^+(e')$. Note that the split of a reflex edge performed to resolve (b) can be unnecessary if e cannot be seen from e'. However, such a split does not introduce an additional convex piece. The upper walls can be handled in the same way as the lower walls.

In the second step of the decomposition, no wall intersects a reflex edge or introduces new reflex edges. Also there are no mutual and cyclic dependencies. Since we use multiple directions, the supporting planes usually intersect and therefore the decomposition is not unique. But the number of sub-polyhedra is constant among all orders of wall creation.

Like the decomposition of Chazelle [4], our decomposition clearly yields at most $O(r^2)$ polyhedra. The wall of a non-vertical reflex edge e can intersect with each other reflex edge at most once. This does not change if other walls cut e into multiple parts, because all sub-walls have the same supporting plane as $W(e)$. Therefore the first step generates at most $O(r^2)$ cylindrical cells and $O(r^2)$ vertical reflex edges. Then, each vertical reflex edge exactly splits one cell

into two. The worst case runtime of our implementation is worse than Chazelle's since we use kd-tree based ray shooting for the insertion of the vertical edges and for the walk along the boundary.

3 The Minkowski Sum of Convex Polyhedra

The Minkowski sum of two convex polyhedra is also a convex polyhedron. Furthermore, it is well known that each vertex $v_{P\oplus Q}$ of the Minkowski sum $P\oplus Q$ is the vector sum of vertices v_P in P and v_Q in Q [15]. Hence, a trivial solution for the Minkowski sum of two convex polyhedra P and Q computes the convex hull of all vector sums of vertex pairs of P and Q. This algorithm performs a convex hull computation on pq vertices, where p and q are the number of vertices in P and Q. Thus, using CGAL's `convex_hull_3` function the trivial algorithm runs in $O(pq\log(pq))$ time.

A more efficient solution can be obtained by using normal diagrams. Each convex polyhedron P has a unique dual representation N_P called the *Gaussian diagram* or *normal diagram*. It is a subdivision of the sphere into vertices, edges and faces, such that the outward-directed normal directions of all planes supporting some item of P constitute an item of N_P. For a facet of P there is exactly one plane supporting it. Thus, its dual item is the single point on the sphere with the same normal direction as the supporting plane. The normal directions of the planes supporting an edge e_P of P form a great arc on the sphere. The endpoints of the great arc are dual items of the facets incident to e_P. A face f_n on N_P is the dual item of a vertex v_p of P. f_n is bound by a convex cycle of edges and vertices, which are the dual items of the edges and facets incident to v_p. The order of the edges and vertices around f_n coincides with the order of dual items around v_p.

The faces of $N_{P\oplus Q}$ are intersections of faces of N_P and N_Q. What is more, the dual face of $v_{P\oplus Q}$ is the intersection of the dual faces of v_P and v_Q with $v_P + v_Q = v_{P\oplus Q}$. As a consequence, the overlay of N_P and N_Q is the normal diagram of the Minkowski sum $P\oplus Q$. Also, the exact point set of $P\oplus Q$ can easily be obtained by storing the primal vertices with their respective dual face and computing the vector sums for each face in the overlay. Thus, using the overlay of normal diagrams improves on the trivial algorithm in two points. First, the construction of $P\oplus Q$ operates on the exact set of vertices, which might be far smaller than pq. However, in the worst case, $P\oplus Q$ still has $O(pq)$ vertices. And second, the incidence structure of $N_{P\oplus Q}$ allows us to construct $P\oplus Q$ from it in time linear to $P\oplus Q$.

With Nef polyhedra embedded on the sphere [11] as provided by CGAL, we already have a tool that can be used to realize normal diagrams, and for which we also have an overlay algorithm that can be reused for the Minkowski sum. The overlay algorithm also allows to store arbitrary data with each vertex, edge, and face, and to propagate this data properly during the overlay. Therefore, the missing operations are the two conversions between a convex three-dimensional polyhedron and its normal diagram. Both functions are easy to implement.

Spherical Nef polyhedra are not the most efficient solution for the overlay of normal diagrams. Asymptotically, there is no essentially superior solution, but the *Cubical Gaussian Map* of Fogel and Halperin is clearly faster [9]. The binary operations on spherical Nef polyhedra can handle more complex overlays than those of normal diagrams, which are always convex arrangements; they never include nested faces or lower dimensional features. Therefore, our overlay algorithm is obviously more costly than needed. Apart from that, the spherical predicates are too expensive.

For the runtime experiments of this paper, the spherical Nef polyhedra are sufficient, since the runtime of computing the Minkowski sum of the convex parts is always much smaller than the union of these partial solutions. For a future release of our implementation of non-convex Minkowski sum computations, we plan to exploit other efficient implementations of algorithms that compute Minkowski sums of convex polyhedra such as the one based on Cubical Gaussian Maps [9], or the one based on Arrangement on Surfaces [3].

4 Uniting a Set of Polyhedra

The union of the Minkowski sums of the convex sub-polyhedra is done by multiple binary union operations of 3D Nef polyhedra. Since the complexity of the binary union operation depends on the complexities of both input and the result polyhedron in equal shares, it is essential not to perform the binary operations in arbitrary order. The trivial method for instance maintains one Nef polyhedron holding the current intermediate result. It starts with an empty polyhedron and adds the polyhedra one by one. This method performs very badly, since most of the union operations involve at least one big polyhedron, namely the intermediate result. Experiments showed that examples that can be computed in less than 10 minutes with efficient methods, run for more than a day with the trivial approach. Clever methods unite small polyhedra first, and try to keep intermediate results as small as possible. Since we cannot foresee the optimal order, we develop and test different strategies.

Our first method performs in a greedy fashion. It maintains the set of all unhandled primitives and intermediate results, and unites the two smallest polyhedra in each step. This can be realised by a priority queue. The priority of a polyhedron is its size measured by the number of its vertices. The priority queue is initialized with all pairwise Minkowski sums of the convex pieces. Then, repeatedly the two smallest polyhedra are extracted from the queue, and their union is inserted into the queue. The method terminates with the result left as the final remaining element in the queue.

Our second strategy tries to unite neighboring polyhedra to reduce the size of the intermediate results. For this purpose, we put the primitives into a queue and sort the queue by the lexicographically smallest vertex of the primitives. In order to unite polyhedra of the same complexity, we proceed similar to the priority queue approach. We extract and unite the first two polyhedra, and append their result to the end of the queue. The neighboring relation used for the sorting is

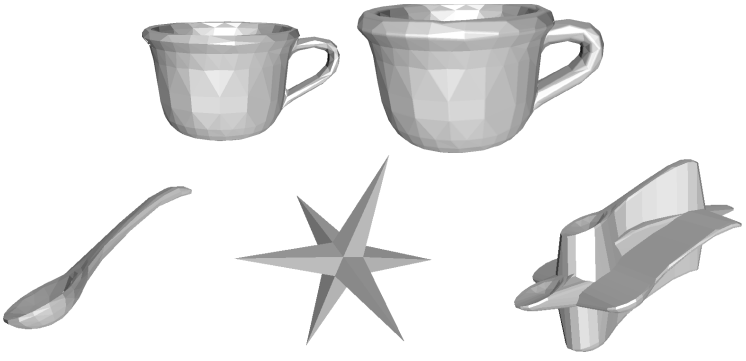


Fig. 4. Example Minkowski sums. Top: cup and cup $\oplus$ sphere. Bottom: spoon, star, and spoon $\oplus$ star.

Table 1. Models used in the experiments

	cube	ball1	ball2	star	spoon	mushroom	cup
facets	6	128	1000	24	336	448	1000
parts	1	1	1	5	186	255	774

maintained throughout the whole union process. Sorting the polyhedra by their lexicographically smallest vertex does not specify how we compare two such vertices in the sorting function. We test three comparison types: lexicographical comparison of the coordinates, comparison of the L1 distance from a point in a corner of the scenery, and comparison of the L2 distance from the origin.

Performing larger examples, it becomes obvious that memory is major issue in either of the above strategies. The queue-based approach can be adapted, such that no more than $\frac{\log p}{2}$ need to be stored, where p is the number of primitives. Instead of a queue we maintain a stack. The primitives are computed and inserted to the stack one by one. After pushing the ith primitive onto the stack, we $\lfloor \log i \rfloor$ times pop and unite the respective top two items and push the result back on the stack. Note that every binary union (besides the ones after the insertion of the final primitive) combines two polyhedra that are unions of the same number of primitives. Although we construct primitives just before they are pushed on the stack, we also want to sort the set of primitives in advance. For this purpose, we additionally store all normal diagrams and a sorted list of all ordered pairs of pointers to the normal diagrams that schedules the creation of the primitives. The list is sorted by the sum of the smallest vertices of the respective sub-polyhedra. Again we test the same three comparison types.

Table 2 summarizes the tests of the strategies. The stack strategy proved to be superior to the others. This becomes clearer the larger the examples get. The difference between the comparison types is very small. Lexicographical comparisons show the best result. The last line of the table shows the runtime of a much bigger example.

Table 2. Performance of the different union strategies, the decomposition and the Minkowski sum of the convex pieces. For each model the number of facets and the number of convex sub-polyhedra are listed. The runtimes are given in seconds.

model1	model2	priority queue	queue			stack			convex decomp	convex sum
			lexi	L1	L2	lexi	L1	L2		
mushroom	cube	170	151	151	152	147	147	149	43	14
mushroom	ball1	608	529	534	545	408	415	423	43	102
spoon	star	963	930	921	925	755	726	732	43	84
cup	ball2					8497			415	939

5 Tight Passages

Computing the Minkowski sum of a tight passage scenario requires the handling of open polyhedra. A Nef polyhedron stores set selections marks for every vertex, edge, facet, and volume, which indicate whether the respective item is part of the polyhedron. In case of an open polyhedron the marks of the boundary items are unselected. These selection marks can additionally be stored in the normal diagram. During the overlay operation the marks are transferred in the following way. Given normal diagrams N_P and N_Q, we consider intersecting items i_P and i_Q. Their intersection forms item $i_{P\oplus Q}$ in the overlay $N_P \oplus N_Q$. Then, the selection mark stored with $i_{P\oplus Q}$ can be computed as $i_P \wedge i_Q$. This means for instance, that a vertex $v_{P\oplus Q} = v_P + v_Q$ of $P \oplus Q$ is selected iff v_P and v_Q are selected.

If we allow arbitrary selection mark for the input, i.e., each boundary part may have a different mark, the computation of the Minkowski sum becomes more complicated. The reason is, that each side of a convex sub-polyhedron may consist of several facets, some of which are boundary parts of the input and some of which are walls inserted by the decomposition. The selection marks of these facets can differ. As a consequence, the selection marks cannot be handled as described above.

Fortunately, we don't need arbitrary selection marks to handle tight passages. It is only necessary to allow polyhedra that are either open or closed, i.e., all selection marks of boundary items are either selected or unselected. In this situation it suffices to ignore the selection marks of the walls. If the side of a sub-polyhedron consists of original facets and walls, we use the uniform mark of the original facets; if there are only walls, we can assign an arbitrary mark. Naturally, the walls must be selected because they represent a point set inside a polyhedron. Thus, not selecting a wall yields an unselected facet in a convex Minkowski sum pr_1 that should be selected. On the other hand, this wall is adjacent to another sub-polyhedron. Because of the adjacency and because of the convexity of primitives, the convex Minkowski sums computed on this other sub-polyhedron must generate a primitive pr_2 that overlaps pr_1 such that the wrongly unselected facet is in the interior of pr_1. The final result of the Minkowski sum operation will not contain any of these facets.

Figure 5 shows a tight passage scenario solved with our implementation.

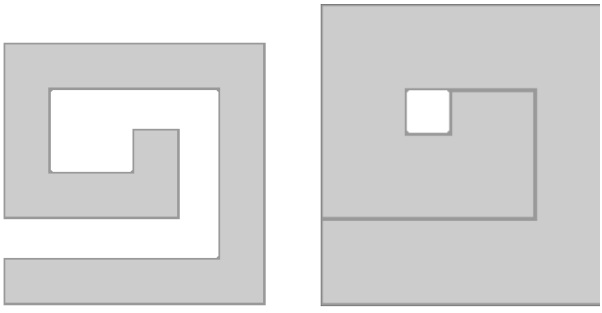


Fig. 5. A maze whose corridors have unit width, and the Minkowski sum of a unit cube and the maze

6 Conclusion

We have presented an exact implementation of the Minkowski sum of non-convex 3D polyhedra. Our implementation also supports open and closed polyhedra, which allows to solve extreme scenarios like tight passage problems.

As part of our Minkowski sum, we have also presented the first robust decomposition of non-convex 3D polyhedra into $O(r^2)$ convex pieces, where r is the number of reflex edges.

As part of future work, we want to further improve the efficiency of our implementation and release it as part of CGAL. We plan two major points of improvement. First, we want to improve the second step of the decomposition by adapting polygon decomposition methods for the decomposition of the cylindrical cells. Those methods would sometimes resolve two vertical reflex edges with one wall and therefore generate fewer convex pieces. As a second step, we plan to replace our solution for the convex Minkowski sums by a faster approach.

Acknowledgements

The author likes to thank Lutz Kettner for stimulating the work of this paper. I thank Efi Fogel for helping me repeat the comparison of theirs and out implementation of the Minkowski sum on convex polyhedra. Furthermore, I thank Liangjun Zhang and Dinesh Manocha for providing 3D models, such that I could repeat two of the experiments in [19]. Also, I would like to thank Mark de Berg for valuable discussions on the presented topic.

References

1. Agarwal, P.K., Flato, E., Halperin, D.: Polygon decomposition for efficient construction of Minkowski sums. Comp. Geom.: Theory and Appl. 21, 39–61 (2002)
2. Aronov, B., Sharir, M.: Triangles in space or building (and analyzing) castles in the air. In: SCG '88: Proceedings of the fourth annual symposium on Computational geometry, pp. 381–391. ACM Press, New York, USA (1988)

3. Berberich, E., Fogel, E., Halperin, D., Wein, R.: Sweeping over curves and maintaining two-dimensional arrangements on surfaces. In: Proc. 23rd Workshop on Computational Geometry (EWCG'07), pp. 223–226 (2007)
4. Chazelle, B.: Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. SIAM Journal on Computing 13(3), 488–507 (1984)
5. Chazelle, B., Dobkin, D., Shouraboura, N., Tal, A.: Strategies for polyhedral surface decomposition: an experimental study. Comput. Geom. Theory Appl. 7(5-6), 327–342 (1997)
6. de Berg, M., Guibas, L.J., Halperin, D.: Vertical decompositions for triangles in 3-space. In: SCG '94: Proceedings of the tenth annual symposium on Computational geometry, pp. 1–10. ACM Press, New York, USA (1994)
7. Ehmann, S.A., Lin, M.C.: Accurate and fast proximity queries between polyhedra using convex surface decomposition. In: Comp. Graph. Forum (Proc. Eurographics 2001), vol. 20(3), pp. 500–510 (2001)
8. Elber, G., Kim, M.-S.: Special Issue of Computer Aided Design: Offsets, Sweeps and Minkowski Sums, vol. 31 (1999)
9. Fogel, E., Halperin, D.: Exact and efficient construction of Minkowski sums of convex polyhedra with applications. In: 7th Workshop on Algorithm Engineering and Experiments (ALENEX 06) (to appear 2006)
10. Guibas, L.J., Ramshaw, L., Stolfi, J.: A kinetic framework for computational geometry. In: 24th Symp. on Found. of Comp. Sci (FOCS), pp. 100–111 (1983)
11. Hachenberger, P., Kettner, L., Mehlhorn, K.: Boolean operations on 3D selective Nef complexes: Data structure, algorithms, optimized implementation and experiments. Comp. Geometry: Theory and Applications 38(1–2), 64–99 (2007)
12. Joe, B.: A software package for the generation of meshes using geometric algorithms. Advances in Engineering Software and Workstations 13(5–6), 325–331 (1991)
13. Kaul, A., Rossignac, J.: Solid-interpolating deformations: Construction and animation of pips. Computers & Graphics 16(1), 107–115 (1992)
14. Kim, Y.J., O., M.A., Lin, M.C., Manocha, D.: Fast penetration depth computation using rasterization hardware and hierarchical refinement. In: Proc. of Workshop on Algorithmic Foundations of Robotics (2002)
15. Latombe, J.-C.: Robot Motion Planning. Kluwer Academic Publishers, Norwell, MA, USA (1991)
16. O'Rourke, J.: Art Gallery Theorems and Algorithms. Oxford University Press, Inc., New York, USA (1987)
17. Rossignac, J.R., Requicha, A. A.G.: Offsetting operations in solid modelling. Comput. Aided Geom. Des. 3(2), 129–148 (1986)
18. Rössl, C., Kobbelt, L., Seidel, H.-P.: Extraction of feature lines on triangulated surfaces using morphological operators. In: Proc. of the 2000 AAAI Symp. (2000)
19. Varadhan, G., Manocha, D.: Accurate Minkowski sum approximation of polyhedral models. In: Proc. Comp. Graphics and Appl., 12th Pacific Conf. on (PG'04), pp. 392–401. IEEE Computer Society Press, Los Alamitos (2004)
20. Wein, R.: Exact and efficient construction of planar Minkowski sums using the convolution method. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 829–840. Springer, Heidelberg (2006)

A New ILP Formulation for 2-Root-Connected Prize-Collecting Steiner Networks

Markus Chimani, Maria Kandyba*, and Petra Mutzel**

Department of Computer Science, University of Dortmund, Germany
{markus.chimani,maria.kandyba,petra.mutzel}@cs.uni-dortmund.de

Abstract. We consider the real-world problem of extending a given infrastructure network in order to connect new customers. By representing the infrastructure by a single root node, this problem can be formulated as a 2-root-connected prize-collecting Steiner network problem in which certain customer nodes require two node-disjoint paths to the root, and other customers only a simple path. Herein, we present a novel ILP approach to solve this problem to optimality based on directed cuts. This formulation becomes possible by exploiting a certain *orientability* of the given graph. To our knowledge, this is the first time that such an argument is used for a problem with node-disjointness constraints. We prove that this formulation is stronger than the well-known undirected cut approach. Our experiments show its efficiency over the other formulations presented for this problem, i.e., the undirected cut approach and a formulation based on multi-commodity flow.

1 Introduction

Extending already existing fiber-optics networks by connecting new customers is an important topic in the design of telecommunication networks. Thereby, we have an existing infrastructure network I, a set of potential new customers C and a set of potential new route-segments for laying the fiber cables. As each new customer v will generate a certain assessable profit $p(v) \in \mathbb{R}^+$ and each route-segment e has a certain laying cost $c(v) \in \mathbb{R}^+$, the main task is to connect a subset of C with I such that the overall profit is maximized. In this paper we consider the real world problem [1], where some of the customers, if added to the network, require two node-disjoint connections to I to increase reliability. We denote these customers with the set C_2, and the other customers with C_1.

By representing the infrastructure network by a single root node r (for the details of such transformation see, e.g., [2]), we obtain a rooted Prize-Collecting Steiner Network problem where certain nodes are required to be (nodewise) 2-connected with the root. Formally, we are given an undirected graph

* Supported by the German Research Foundation (DFG) through the Collaborative Research Center "Computational Intelligence" (SFB 531).
** Partially supported by the Austrian Research Promotion Agency (FFG) under grant 811378 (NetQuest project).

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 681–692, 2007.

$G = (V, E)$, a root node $r \in V$, a set of customer nodes $C = C_1 \dot\cup C_2 \subset V$, a prize function $p : V \to \mathbb{R}^+$, and a cost function $c : E \to \mathbb{R}^+$. Find a subgraph $N = (V_N, E_N)$ of G with $r \in V_N$ which minimizes $\sum_{e \in E_N} c(e) - \sum_{v \in V_N} p(v)$ and satisfies the following connectivity property: for every node $v \in C_k \cap V_N$ ($k \in \{1, 2\}$), N contains at least k node-disjoint paths connecting v to r.

We call such a problem a *2-Root-connected Prize-Collecting Steiner Network* problem (2RPCSN). If we require all customers to be included into the solution network, the resulting problem is called *2-Root-connected Steiner Network* problem (2RSN). Both 2RPCSN and 2RSN are NP-hard, as they contain the Steiner tree problem as a special case. While our paper centers on the investigation of 2RPCSN, all results clearly also hold for 2RSN. Furthermore, our approach can be used for the relaxed version where C_2 customers are only required to be 2-edge-connected with the root.

2RPCSN was already studied in [18,19], where two different ILP formulations for this problem were suggested: one based on multi-commodity flow, similar to [13], the other one using undirected cut inequalities[1]. In this paper, we tranform 2RPCSN into the problem of finding an optimal subgraph in a related directed graph and give a new ILP formulation which uses directed cut inequalities. To our knowledge, our formulation is the first which applies such an approach to a node-disjoint connectivity problem, cf. Section 1.1. Furthermore, we study the polyhedral properties of our ILP and show that our formulation is stronger than the undirected cut formulation. We solve 2RPCSN using this new formulation within a Branch-and-Cut framework, utilizing an LP-based heuristic also presented herein. Our experimental results in Section 3 show that our approach is superior to those of [18,19] for nearly all test instances.

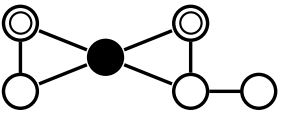
1.1 Basics and Related Work

For a set $W \subset V$ of an undirected graph $G = (V, E)$ we denote the set of edges which separate W from $V \setminus W$ by $\delta_G(W) = \{\{u, v\} \in E \mid u \in W, v \in V \setminus W\}$. For a directed graph G', we distinguish between $\delta^-_{G'}(W)$ and $\delta^+_{G'}(W)$, i.e., the set of cut edges having a source or a target node in W, respectively. We may drop the index specifying the graph, if the graph is clear from the context.

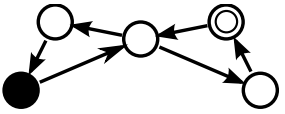
2RSN is a special class of survivable network design problems (SNDP) [16]; see, e.g., [9,20] for surveys. In [17] the following variant of SNDP is considered: each node v of the given graph is associated with a nonnegative integer $r_v \in \{0, \dots, k\}$. For each pair of nodes $u, v \in V$ the connectivity requirement is then defined by $r_{uv} = \min\{r_u, r_v\}$, i.e., there should be at least r_{uv} edge- or node-disjoint paths between those nodes. These problems are called kECON and kNCON, respectively. In general, 2RSN and 2NCON are not equivalent, cf. Fig. 1(a).

For kECON and kNCON, Grötschel, Monma, and Stoer [7] described integer linear programs and investigated their polyhedral structure. The central idea

[1] Although the paper's title uses the term "directed cut", it turns out to be equivalent to the traditional undirected approach discussed in Section 2.3.



(a) This network is infeasible for 2NCON, but feasible for 2RSN.



(b) This orientation satisfies the properties of Robbins' proof, but there are no two node-disjoint paths between the root and the C_2 customer.

Fig. 1. The root node is denoted by the black circle, C_1 and C_2 customers are denoted by simple and double circles, respectively

is to express the connectivity requirements by undirected cuts: for every non-empty set of nodes $W \subset V$ the number of edges in $\delta(W)$ should be at least $max\{r_{uv} \mid u \in W, v \in V \setminus W\}$. Wagner et al. [19] formulated their ILP for 2RSN and 2RPCSN using basically the same idea.

An *orientation* of an undirected graph G is a directed graph G' which is obtained by transforming each edge of G into a directed edge. Robbins [15] (for the special case of $k = 1$) and Nash-Williams [14] showed that for any graph G there exists an orientation G' with the following property: for every pair of nodes u, v which is $2k$-connected in G, there exist k pairwise edge-disjoint directed paths $(u \to v)$ and k pairwise edge-disjoint directed paths $(v \to u)$ in G'.

This fact has been exploited by Chopra [5] for solving 2ECON via directed graphs, who proved his formulation to be superior to the undirected formulation. Goemans [6] and Stoer [17] extended this formulation to kECON for the case that all connectivity requirements are 0, 1, or even; later Magnanti and Raghavan [13] extended it for general k. It has been an open problem [17] if a similar orientation technique can be used for kNCON-type problems, i.e., when we require node-disjointness. Considering 2RSN and 2RPCSN, where we require nodewise 2-connectedness with a special root node, we will show that this is indeed the case.

Note that the above described approaches considered Steiner networks, and did not consider the prize-collecting variants. If $C_2 = \emptyset$, 2RPCSN becomes a Rooted Prize-Collecting Steiner Tree (RPCST) problem which has been investigated, e.g., by [10,11]. Therein, the authors presented several ILP models, including a directed cut approach, which turned out to be the most successful one.

2 Investigating 2RPCSN

2.1 Transformation into a Directed Problem

The central idea of our formulation is that we can transform the undirected 2RPCSN problem $(G=(V,E), r, c, p)$ into a directed variant $(G' = (V, A), r, c', p)$ as follows: for each edge $\{u, v\} \in E$ there are two directed edges (u, v) and (v, u) in A, with $c'((u, v)) = c'((v, u)) = c(\{u, v\})$. An optimal solution of this directed

problem (D2RPCSN) is a subgraph $D = (V_D \subseteq V, A_D \subseteq A)$ with $r \in V_D$, which minimizes $\sum_{e \in A_D} c'(e) - \sum_{v \in V_D} p(v)$ and satisfies:

(D1) For each edge $\{u, v\} \in E$, A_D may include at most one of the arcs (u, v) and (v, u).
(D2) For each customer node $v \in C_1 \cap V_D$ there is a directed path $(r \to v)$ in D.
(D3) For each customer node $v \in C_2 \cap V_D$ there is a path $(r \to v)$ and a path $(v \to r)$ in D, which are node-disjoint.

We show that D2RPCSN is equivalent to 2RPCSN. Clearly, every feasible solution of D2RPCSN can be interpreted as a feasible solution of 2RPCSN with the same objective value straightforwardly. Hence we have to focus on the reverse transformation from 2RPCSN to D2RPCSN. As we can see for the single C_2 node in Figure 1(b), the existence of node-disjoint paths does not follow from the orientability theorem [15] exploited by Chopra [5]. However note that, in this example, a reorientation of the 3-cycle containing this customer would result in a valid orientation for D2RPCSN.

Theorem 1. *Any optimal solution for 2RPCSN can be transformed into a corresponding feasible solution for D2RPCSN with the same objective value.*

Proof. Let $N = (V_N, E_N)$ be an optimal solution for 2RPCSN. We will show that there exists an orientation D of N which is a feasible solution for the corresponding D2RPCSN problem.

We first shrink N by orienting all *attached trees*, i.e., we iteratively find an edge $\{u, v\}$ where v has degree 1; we orient the edge from u to v, and remove it temporarily. The remaining graph structure of undirected edges consists of one or more at least 2-connected components attached to r. It is clear that by orienting each one of them separately, we obtain a valid orientation for the complete structure. Hence we can restrict ourselves to a 2-connected undirected graph $B = (V_B, E_B)$ which contains r.

We use $\ell : V_B \to \mathbb{R} \cup \{\textit{undefined}\}$ as a labeling function; initially we have $\ell(v) := \textit{undefined}$ for all $v \in V_B$. We start by identifying a simple cycle Z in B containing r, and orient its edges consistently in one of the two possible directions. We then label each node on Z with increasing fractional numbers between 0 and 1, according to this orientation, starting with $\ell(r) := 0$. Hence, all edges of Z (except its last edge $\hat{e}$) are oriented from the smaller towards the larger label number. We will now orient the remaining undirected edges in such a way that this invariant is valid for all oriented edges:

We define an *augmenting path* $P = (a \to b)$ as a simple path of unoriented edges where only the disjoint start and end nodes are labeled, and $\ell(a) < \ell(b)$.

To orient B, we repeatedly find an augmenting path $P = (a \to b)$ and orient it from a to b, labeling all inner nodes with increasing numbers greater than $\ell(a)$ but smaller than $\ell(b)$; these labels are to be unique over all labelings so far. By this construction, we guarantee that each labeled node has at least one incoming and one outgoing edge. Furthermore, each oriented edge is oriented

from the smaller towards the larger label number. Hence, each oriented path will always contain monotonously increasing label numbers (with the exception of $\hat{e}$). This means that any directed circle starting from r and going through any labeled node v will be simple, and we therefore have node-disjoint paths $(r \to v)$ and $(v \to r)$.

It remains to show that every edge gets oriented by this process. Assume that at some point there is at least one unoriented edge e left, but we cannot find any augmenting path. Clearly, e has to be part of some shortest path $Q = (c \to d)$ of unoriented edges with labeled nodes c and d. Since neither Q nor its reversal is an augmenting path, we have $\ell(c) = \ell(d)$ and therefore $c = d$, i.e., Q is a cycle of unoriented edges, and none of its nodes except for c are labeled. Since B is 2-connected, there has to be an additional unoriented path from some node $q \in Q$ to some labeled node p ($p, q \neq c$). But then, the path $(p \to q \to c)$ (or its reversal) would be an augmenting path, which is a contradiction. Hence, the above algorithm correctly orients any optimal solution of 2RPCSN. □

2.2 ILP for D2RPCSN

To model D2RPCSN on G' we introduce two sets of binary variables

$$x_e, y_v \in \{0, 1\} \quad \forall e \in A, \forall v \in V.$$

The variables are 1, if the corresponding node or edge is in the solution network D, and 0 otherwise. We therefore obtain the objective funtion:

$$\min \sum_{e \in A} c'(e) \cdot x_e - \sum_{v \in V} p(v) \cdot y_v. \tag{1}$$

In a feasible solution of D2RPCSN, at most one of the edges corresponding to an undirected edge $\{u, v\}$ can be selected. Furthermore, selecting an edge requires both incident nodes to be selected as well:

$$x_{uv} + x_{vu} \leq y_v \qquad \forall v \in V, \forall (v, u) \in A. \tag{2}$$

The *forward-cut* constraints are traditional cut-constraints requiring the selected customer nodes to be reachable from the root.

$$\sum_{e \in \delta^+(S)} x_e \geq y_v \qquad \forall S \subseteq V \setminus \{r\}, \forall v \in S \cap C. \tag{3}$$

For customers requiring 2-connectedness we have analogous *backward-cut* constraints:

$$\sum_{e \in \delta^-(S)} x_e \geq y_v \qquad \forall S \subseteq V \setminus \{r\}, \forall v \in S \cap C_2. \tag{4}$$

Finally, we have to assure that the 2-connected customer nodes are connected via node-disjoint forward- and backward-paths, i.e., we require each node w to be part of at most one of these paths. This is equivalent to require that at least one

of these paths does not contain w. Let G'_w denote the graph G' without the node w and its incident edges. Then we have $\forall S_1, S_2 \subseteq V \setminus \{r\}, \forall v \in S_1 \cap S_2 \cap C_2, \forall w \in V \setminus \{r, v\}$:

$$\sum_{e \in \delta^+_{G'_w}(S_1)} x_e + \sum_{e \in \delta^-_{G'_w}(S_2)} x_e \geq y_v. \tag{5}$$

2.3 Polyhedral Comparison

We compare our ILP formulation to the common and straightforward formulation of 2RPCSN based on the undirected graph and undirected cuts. This formulation was, e.g., used in [19], although in a slightly more redundant form. Thereby, we have the characteristic vector $z \in \{0,1\}^{|E|}$ specifying the selected edges, and the vector y analogous to the definition in Section 2.2. We then have:

$$\min \sum_{e \in E} c(e) \cdot z_e - \sum_{v \in V} p(v) \cdot y_v \tag{6}$$

$$\sum_{e \in \delta(S)} z_e \geq y_v \qquad \forall S \subseteq V \setminus \{r\}, \forall v \in S \cap C_1 \tag{7}$$

$$\sum_{e \in \delta(S)} z_e \geq 2y_v \qquad \forall S \subseteq V \setminus \{r\}, \forall v \in S \cap C_2 \tag{8}$$

$$\sum_{e \in \delta_{G_w}(S)} z_e \geq y_v \qquad \forall S \subseteq V \setminus \{r\}, \forall v \in S \cap C_2, \forall w \in V \setminus \{r, v\} \tag{9}$$

$$z_e, y_v \in \{0,1\} \qquad \forall e \in E, \forall v \in V \tag{10}$$

Let us consider any 2RPCSN problem and its corresponding D2RPCSN counterpart. For corresponding solutions, we clearly have the projection $z_{uv} = x_{uv} + x_{vu}$.

Let $\mathcal{P}_U$ and $\mathcal{P}_D$ be the polyhedrons corresponding to feasible LP relaxations, i.e., feasible solutions for the ILP without integrality constraints, for 2RPCSN and D2RPCSN, respectively. We show that $\mathcal{P}_D \subset \mathcal{P}_U$, i.e., the lower bounds obtained by the LP relaxations of our new formulation will in general be tighter than for the undirected formulation. The proof technique is based on [5,7], but had to be extended for the prize-collecting setting.

Observation 1. $\mathcal{P}_D \neq \mathcal{P}_U$.

Proof. Consider a triangle graph with the root node and two high-profit C_1 customer nodes. For 2RPCSN the fractional solution of 0.5 on all edges will satisfy all edge constraints. For D2RPCSN any solution corresponding to this undirected solution would be infeasible. □

Theorem 2. *The directed cut formulation is stronger than the undirected cut formulation. I.e.,* $\mathcal{P}_D \subset \mathcal{P}_U$.

Proof. Due to Observation 1, it is enough to show $\mathcal{P}_D \subseteq \mathcal{P}_U$. Hence, we have to show that the undirected cut inequalities can be generated from their directed counterparts, based on $z_{uv} = x_{uv} + x_{vu}$.

Consider any set $S \subseteq V \setminus \{r\}$. For $v \in C_2 \cap S$ we have:

$$\sum_{(u,v)\in\delta^+(S)} x_{uv} + \sum_{(v,u)\in\delta^-(S)} x_{vu} = \sum_{\{u,v\}\in\delta(S)} x_{uv} + x_{vu} = \sum_{\{u,v\}\in\delta(S)} z_{uv} \geq 2y_v.$$

For $v \in C_1 \cap S$ we have:

$$\sum_{\{u,v\}\in\delta(S)} z_{uv} = \sum_{\{u,v\}\in\delta(S)} x_{uv} + x_{vu} = \sum_{(u,v)\in\delta^+(S)} x_{uv} + \sum_{(v,u)\in\delta^-(S)} x_{vu} \geq y_v.$$

Analogously, we generate the undirected node-disjointness inequalities for any node $v \in S \cap C_2$ and $w \in V \setminus \{r, v\}$:

$$\sum_{\substack{\{u,v\}\\ \in\delta_{G_w}(S)}} z_{uv} = \sum_{\substack{\{u,v\}\\ \in\delta_{G_w}(S)}} x_{uv} + x_{vu} = \sum_{\substack{(u,v)\\ \in\delta^+_{G'_w}(S)}} x_{uv} + \sum_{\substack{(v,u)\\ \in\delta^-_{G'_w}(S)}} x_{vu} \geq y_v. \qquad \square$$

2.4 Polynomial Separation and Branch-and-Cut

Based on our D2RPCSN ILP formulation, we developed a Branch-and-Cut code. For a general description of the Branch-and-Cut scheme see, e.g., [21]. Generally, such algorithms start with solving the *LP relaxation*, i.e., the ILP without the integrality property, only considering a certain subset of all constraints. Given the *fractional solution* of this partial LP, we perform a *separation routine*, i.e., identify constraints of the full constraint set which the current solution violates. We then add these constraints to our current LP and reiterate these steps. If at some point we cannot find any violated constraints, we have to resort to *branching*, i.e., we generate two disjoint subproblems, e.g., by fixing a variable to 0 or 1. By using the LP relaxation as a lower bound, and our heuristic solution (cf. Section 2.5) as an upper bound, we can prune irrelevant subproblems.

In our case, we start with the constraints (2) and the subset of the constraints (3) for $|S| = 1$. In the optimal solution, the root is the only node which may have only outgoing but no incoming edges. Analogously, no node, except for C_1 customers, will ever have only incoming edges. Although such *flow-preservation* constraints do not strengthen the formulation, adding them to the initial constraint set can help to increase the efficiency of our Branch-and-Cut approach. In our experiments we added all constraints of the second type for the *ClgM* and *ClgM*$^+$ instances, cf. Section 3.

The cut constraints (3) can be separated in polynomial time via the traditional max-flow separation scheme: after obtaining some LP relaxation for our partial ILP, we compute the maximum flow from r to each $v \in C$ in G using the edge values of the current solution as capacities. If the resulting value is less then y_v, we extract one or more of the induced minimum r-v-cuts and add the

corresponding constraint(s) to our ILP model. The cut constraints (4) can be separated analogously.

If there are no violated constraints of type (3) or (4), we solve the separation problem for the constraints of type (5) in an analogous way: for each node $v \in C_2$ and for each node $w \in V$, $w \neq v$ we compute both the v-r and r-v maximal flows in G'_w. If the sum of these flows is less than y_v, we add the corresponding inequalities. Actually, we do not need to perform the separation routine for each node w: let us consider an integer solution where the constraints (3) and (4) are valid, i.e., we have edge-disjoint paths $(r \to v)$ and $(v \to r)$ for any $v \in C_2$. Assume these paths have a common node w, then there are at least two incoming and two outgoing edges at w. More general, this means that in our fractional solution, we have to consider only nodes w satisfying $\sum_{e \in \delta^-(w)} x_e > 1$ and $\sum_{e \in \delta^+(w)} x_e > 1$.

2.5 Primal Heuristic

A fractional solution of an LP relaxation is used to construct a feasible solution, thus obtaining upper bounds for the optimal solution. We proceed in three steps (see [4] for details):

Construct a Steiner tree. Interpreting all customers as C_1 customers, we can use the LP-based heuristic by Ljubic [10] which computes a Steiner tree T based on the values of x and y.

Assure 2-connectivity. We iteratively extend T to assure 2-connectivity with the root: for each customer $v \in C_2$ we temporarily remove the unique path $P_v = (r \to v)$ in the original tree T and compute a shortest path from v to any node already selected in our current solution. By adding this path, we obtain two node-disjoint paths between v and r. Since thereby all nodes of P_v also become 2-connected, we have to perform this operation only for the C_2 customers which do not have any further C_2 customers in their subtrees of T.

Shrinking. In general, S can be further improved by local optimizations. We know that S consists of one or more non-trivial 2-connected components, which have only the root node in common. All other components of the graph form trees, which are attached to some 2-connected component. As described in [21], the rooted PCST problem can be solved in linear time, when applied to trees. We use this algorithm to optimize all attached trees, using the attachment node as its root. For the next step, these root nodes will be treated as C_1 customers with corresponding prizes. For every block B of S we compute its *core graph* $\tilde{B}$, where every chain of edges only containing nodes $v \in V \setminus (C_2 \cup \{r\})$ is replaced by a single edge. We remove an edge e from $\tilde{B}$ if all connectivity requirements are still satisfied for $\tilde{B} - e$. Let P_e be the path in B corresponding to e. We can decide in linear time which edges and C_1 nodes of P_e are worth to connect and which are not, and modify or remove the path accordingly.

Use within Branch-and-Cut. We run this heuristic after every 10th computation of an LP-relaxation. Furthermore, we use the heuristic to generate an initial

solution, choosing $y_v := 1$ for all $v \in C$ and using $c(.)$ as edge costs for the initial Steiner tree T.

3 Experiments

We implemented our Branch-and-Cut algorithm in C++, using CPLEX 9.0 and LEDA 5.0.1. The tests were run on a 2.4 GHz AMD Opteron with 2GB of RAM per process. For the experiments we used three different sets of instances presented in [1], and compared the results of our directed cut (DC) approach with those of the multi-commodity flow (MCF) and undirected cut (UC) formulations, which were partially published in [18,19][2]. As it was the case in these publications, we applied a time limit of 2 hours per problem instance. See Figure 2 for diagrams corresponding to the experiments described below: generally, the vertical axis shows the required CPU time in seconds on a logarithmic scale, whereas the horizontal axis corresponds to the instances (in lexicographical order). When an instance could not be solved to provable optimality, there is no corresponding data point for the according formulation.

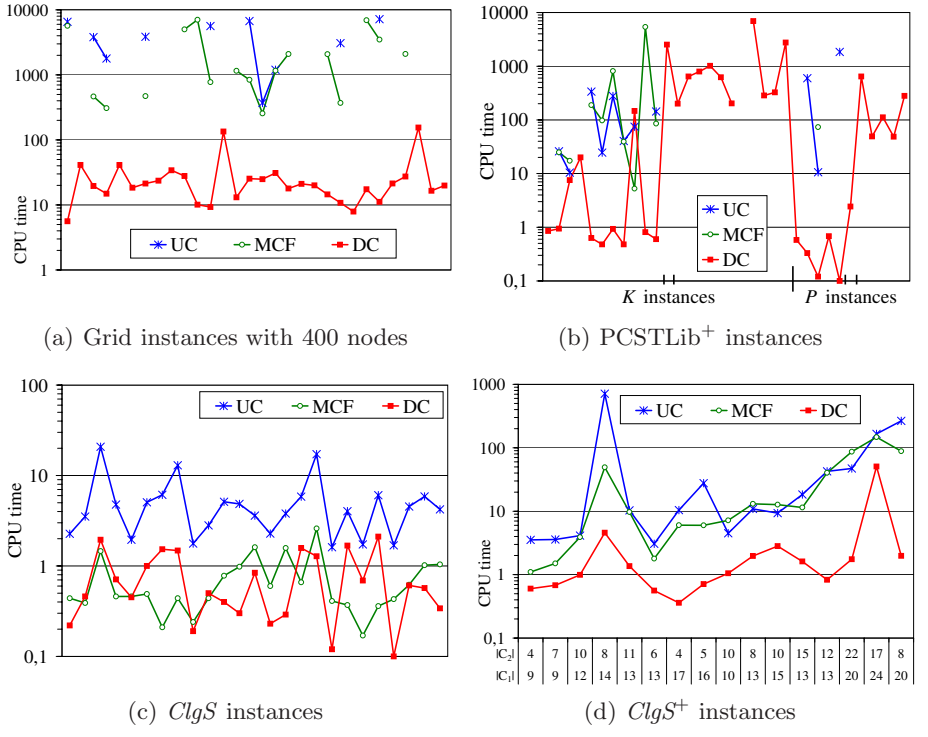
Table 1. Median/average CPU time for the grid instances. The percentage of successful instances is given in brackets if not 100%.

# nodes	100	400	900	1600	2500
UC	9.36/41.3	3834/4021 (33.3%)	(0%)	(0%)	(0%)
MCF	5.15/25.6	1161/2638 (56.7%)	(0%)	(0%)	(0%)
DC	0.10/0.25	20.0/28.6	214/423	1615/1840	1856/2238 (73.3%)

Grid Instances. We used the artificial grid instances with 100, 400, 900, 1600, 2500 nodes [1]. There are two different infrastructure layings I per graph size; for each such I, there are 15 instances with different sets of customer nodes. The instances have 5–13 C_1 and 3–8 C_2 customers. Our algorithm is able to solve all of these instances to optimality, except for 8 instances with 2500 nodes. The largest instances solvable by the previous approaches contained 400 nodes and the running times are much longer, cf. Table 1 and Figure 2(a). Interestingly, our approach required no branching for the instances with 900–2500 nodes.

PCSTLib$^+$*.* This set of instances is based on the instances of PCSTLib [8], also used, e.g., in [3,10,12], and were extended in [1] for 2RPCSN. We used the instances of the groups K and P; each contains graphs with 100, 200, and 400 nodes. The set K consists of random geometric instances which were designed to have a structure similar to street maps. While, for the K instances, 15%–27% percent of the nodes are customers, the P instances have 34%–50% customers; in all instances there are roughly twice as many C_1 nodes as C_2 nodes. As

[2] The algorithms by Wagner et al. were run by Wagner on a stronger Intel Xeon 3.6GHz with CPLEX 10.0.1 and LEDA 5.1.



(a) Grid instances with 400 nodes

(b) PCSTLib$^+$ instances

(c) *ClgS* instances

(d) *ClgS*$^+$ instances

Fig. 2. Diagrams comparing the undirected cut (UC) and the multi-commodity flow (MFC) approach to our directed cut (DC) formulation

shown in Figure 2(b), we solve all P and K instances to optimality, except for a single K instance with 400 nodes. MCF can solve most K instances with 100 nodes, but no larger instances. It solves only a single 100 node P instance, which seems to be due to the high number of customers. This is the only case, where UC is comparable and sometimes even stronger than MCF, but still it is much weaker than DC. These modified PCSTLib instances are the only ones, where DC regularly has to branch, requiring 24 branch nodes on average (the median is 2).

Cologne Instances. These instances use the real-world access net data of the city district Cologne-Ossendorf. For our experiments we took the small (*ClgS*) and medium (*ClgM*) sets of instances [1]: 20 instances with 190 nodes and 377 edges, and 25 instances with 1757 nodes and 3877 edges, respectively. Since these instances have a quite small number of customers (3–6 C_1 and 2–3 C_2 customers), we generated additional sets of instances for both sizes by choosing additional customers, resulting in 16 ClgS$^+$ and 6 ClgM$^+$ instances.

Figure 2(c) gives the results for *ClgS* and shows that MCF and DC are competitive and both clearly outperform the undirected cut approach. As soon as

there are more customers, as it is the case for $ClgS^+$ depicted in Figure 2(d), DC is again superior to two both other approaches.

For the *ClgM* instances with very few customers, MCF is stronger than the others, being able to solve 14 instances, whereas DC solves 9. DC only solves 5 when we run it without the flow-preservation constraints, and UC does not solve any instance. This seems to be due to the fact that in these instances the solutions contain very long paths between the customer nodes, giving the flow formulation an advantage over the cut approaches. This is supported by the observed gain by using the flow-preservation constraints. When there are more customers, which is the case for $ClgM^+$, the average path length is reduced and DC becomes dominant again: while MCF and UC are unable to solve any instance, DC can solve 1. Furthermore, after two hours, DC obtains better lower bounds than MCF and UC in 4 out of 6 cases, and always has a better feasible solution.

4 Additional Remarks

In [18,19], the 2RPCSN problem is also considered with a special relaxation for the C_2 customers: these customers are not required to be 2-connected with the root, but to be *near* to a node with such a property. Therefore, each edge e has a certain *distance* $d(e)$, and each selected C_2 customer v is allowed to have a *connection path* of length up to $k(v)$, if it itself is not 2-connected. Analogously to [19], we can add these constraints to our ILP, using additional y' variables and backward-cuts to decide whether a (non-C_2) node is 2-connected. Anyhow, we recommend to only start with the constraints

$$\sum_{w \in N(v)} y'(w) \geq y(v) \qquad \forall v \in C_2 \tag{11}$$

where $N(v)$ is the set of nodes in the neighborhood of v, i.e., there exists a path of length at most $k(v)$ to each of them in G. Although this constraint is not sufficient, it allows us to introduce a column-generation scheme, where the stricter constraints and corresponding variables for a C_2 node are only introduced if the above constraint does not lead to a short enough connection path.

Acknowledgments. We are deeply indepted to Ivana Ljubic for gratefully allowing us to use her PCST Branch-and-Cut code [10] as a basis of our implementation and for helpful discussions. We also thank Daniel Wagner for conducting additional experiments with his undirected cut and multi-commodity flow implementations [18,19] for our comparison, and the anonymous reviewer for a hint to simplify our orientability proof.

References

1. Bachhiesl, P.: The OPT- and the SST-problems for real world access network design – basic definitions and test instances. Working Report NetQuest 01/2005, Carinthia Tech Institute, Klagenfurt, Austria (2005)

2. Bachhiesl, P.: The OPT- standard problem, solvers, and results. Working Report NetQuest 02/2005, Carinthia Tech Institute, Klagenfurt, Austria (2005)
3. Canuto, S.A., Resende, M.G.C., Ribeiro, C.C.: Local search with perturbations for the prize-collecting steiner tree problem in graphs. Networks 38(1), 50–58 (2001)
4. Chimani, M., Kandyba, M., Mutzel, P.: A new ILP formulation for a 2-connected prize collecting steiner network problem (tr). Technical Report TR07-1-001, Chair for Algorithm Engineering, Dep. of CS, University Dortmund (2007)
5. Chopra, S.: The equivalent subgraph and directed cut polyhedra on series-parallel graphs. SIAM J. Discrete Math. 5(4), 475–490 (1992)
6. Goemans, M.X.: Analysis of linear programming relaxations for a class of connectivity problems. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (1990)
7. Grötschel, M., Monma, C.L., Stoer, M.: Polyhedral Approaches to Network Survivability. In: Reliability of Computer and Communication Networks, Proc. Workshop 1989. Discrete Mathematics and Theoretical Computer Science, vol. 5, pp. 121–141. American Mathematical Society (1991)
8. Johnson, D.S., Minkoff, M., Phillips, S.: The prize-collecting steiner tree problem: Theory and practice. In: Proceedings of 11th ACM-SIAM Symposium on Distcrete Algorithms, San Fransisco, CA, pp. 760–769 (2000)
9. Kerivin, H., Mahjoub, A.R.: Design of survivable networks: A survey. networks 46(1), 1–21 (2005)
10. Ljubic, I.: Exact and Memetic Algorithms for Two Network Design Problems. PhD thesis, Technische Universität Wien (2004)
11. Ljubic, I., Weiskircher, R., Pferschy, U., Klau, G., Mutzel, P., Fischetti, M.: An algorithmic framework for the exact solution of the prize-collecting steiner tree problem. Mathematical Programming, Series B 105(2–3), 427–449 (2006)
12. Lucena, A., Resende, M.G.C.: Strong lower bounds for the prize-collecting steiner problem in graphs. Discrete Applied Mathematics 141(1-3), 277–294 (2003)
13. Magnanti, T.L., Raghavan, S.: Strong formulations for network design problems with connectivity requirements. Networks 45(2), 61–79 (2005)
14. Nash-Williams, C.: On orientations, connectivity and odd-vertex pairings in finite graphs. Canad. J. Math. 12, 555–567 (1960)
15. Robbins, H.E.: A theorem on graphs with an application to a problem of traffic control. American Mathematical Monthly 46, 281–283 (1939)
16. Steiglitz, K., Weigner, P., Kleitman, D.J.: The design of minimum-cost survivable networks. IEEE Trans Circuit Theory 16, 455–460 (1969)
17. Stoer, M.: Design of Survivable Networks. Lecture Notes in Mathematics, vol. 1531. Springer, Heidelberg (1992)
18. Wagner, D., Raidl, G.R., Pferschy, U., Mutzel, P., Bachhiesl, P.: A multicommodity flow approach for the design of the last mile in real-world fiber optic networks. In: Operations Research Proceedings 2006, Springer, Heidelberg (2006)
19. Wagner, D., Raidl, G.R., Pferschy, U., Mutzel, P., Bachhiesl, P.: A Directed Cut for the Design of the Last Mile in Real-World Fiber Optic Networks. In: Proceedings of the International Network Optimization Conference 2007 (2007)
20. Winter, P.: Steiner problem in networks: A survey. networks 17(2), 129–167 (1987)
21. Wolsey, L.A.: Integer Programming. Wiley-Interscience, New York, USA (1998)

Algorithms to Separate $\{0, \frac{1}{2}\}$-Chvátal-Gomory Cuts*

Arie M.C.A. Koster[1], Adrian Zymolka[2], and Manuel Kutschka[3]

[1] University of Warwick, Centre for Discrete Mathematics and its Applications (DIMAP), Coventry CV4 7AL, United Kingdom
Arie.Koster@wbs.ac.uk
[2] atesio GmbH, Sophie-Taeuber-Arp-Weg 27, D-12205 Berlin, Germany
zymolka@atesio.de
[3] Zuse Institute Berlin (ZIB), Takustr. 7, D-14195 Berlin, Germany
kutschka@zib.de

Abstract. Chvátal-Gomory cuts are among the most well-known classes of cutting planes for general integer linear programs (ILPs). In case the constraint multipliers are either 0 or $\frac{1}{2}$, such cuts are known as $\{0, \frac{1}{2}\}$-cuts. It has been proven by Caprara and Fischetti [7] that separation of $\{0, \frac{1}{2}\}$-cuts is $\mathcal{NP}$-hard.

In this paper, we study ways to separate $\{0, \frac{1}{2}\}$-cuts effectively in practice. We propose a range of preprocessing rules to reduce the size of the separation problem. The core of the preprocessing builds a Gaussian elimination-like procedure. To separate the most violated $\{0, \frac{1}{2}\}$-cut, we formulate the (reduced) problem as integer linear program and develop some simple heuristic separation routines.

Computational experiments on benchmark instances show that the combination of preprocessing with exact and/or heuristic separation is a very vital idea to generate strong generic cutting planes for integer linear programs and to reduce the overall computation times of state-of-the-art ILP-solvers.

1 Introduction

Each pure integer linear program (ILP) can be written in its standard minimization form

$$\begin{cases} \min c^T x \\ s.t.\ Ax \leq b \\ \quad x \geq 0 \\ \quad x \in \mathbb{Z}^n \end{cases} \tag{1}$$

with integer matrix $A \in \mathbb{Z}^{m \times n}$, an integer right hand side $b \in \mathbb{Z}^m$, and arbitrary objective values $c \in \mathbb{R}^n$ (here m is the number of rows and n the number of columns of A). Upper bound constraints for single variables are included in the coefficient matrix.

We assume without loss of generality that each row in A has relatively prime coefficients, since otherwise the row can be simplified by dividing all coefficients and the right hand side with the greatest common divisor among the coefficients (after division,

* Most of the research has been carried out while the first and second author were at Zuse Institute Berlin (ZIB). The first author has been supported by the DFG research group "Algorithms, Structure, Randomness" (Grant number GR 883/9-3, GR 883/9-4) during that time.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 693–704, 2007.

a fractional right hand side can be rounded down). Associated with the program (1), we define the integer solution set $X = \{x \in \mathbb{Z}^n \mid Ax \leq b, x \geq 0\}$, its convex hull polyhedron $P_{IP} = \text{conv}(X)$ and the linear relaxation polyhedron $P_{LP} = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$.

Given a system $Ax \leq b$, an *undominated Chvátal-Gomory* (CG) cut is defined by

$$\lfloor u^T A \rfloor \, x \leq \lfloor u^T b \rfloor \tag{2}$$

with $u \in [0,1)^m$ and $\lfloor . \rfloor$ denotes the component-wise rouding. By the integrality of $x \in X$, (2) is valid for P_{IP}. Gomory [14,15] showed that if $P_{IP} \neq P_{LP}$, there exists for every fractional vertex $x^* \in P_{LP}$ a CG cut (2) that is violated, i.e., $\lfloor u^T A \rfloor \, x^* > \lfloor u^T b \rfloor$, see also [10]. Caprara and Fischetti [7] introduced $\{0, \frac{1}{2}\}$-*cuts* for those CG cuts that are derived by $u \in \{0, \frac{1}{2}\}^m$. For several combinatorial optimization problems it is known that problem-specific classes of facet-defining inequalities are $\{0, \frac{1}{2}\}$-cuts with particular properties, e.g., the blossom inequalities of the matching polytope (describing this polytope completely) [11], (odd-valued) odd hole inequalities of the stable (multi-)set polytope [17,18,20], or the Möbius ladder inequalities of the linear ordering polytope [12].

For $\{0, \frac{1}{2}\}$-cuts, we consider the following separation problem:

$\{0, \frac{1}{2}\}$-SEP
Given: The program (1) and a fractional solution $x^* \in P_{LP}$.
Find: A weight vector $u \in \{0, \frac{1}{2}\}^m$ such that $\lfloor u^T A \rfloor \, x^* > \lfloor u^T b \rfloor$ or a proof that none exists.

Theorem 1 (Caprara and Fischetti [7]). $\{0, \frac{1}{2}\}$-SEP *is* $\mathcal{NP}$*-complete.*

Consequently, Caprara and Fischetti [7] concentrate on polynomial-time solvable cases of $\{0, \frac{1}{2}\}$-SEP. In particular, they show that if A is an integer matrix with at most two odd coefficients per row, $\{0, \frac{1}{2}\}$-SEP is polynomial-time solvable. They propose therefore to weaken A to a matrix with the described property. In [4], a computational study is presented to reveal the strength of this heuristic approach for $\{0, \frac{1}{2}\}$-SEP. They restrict cuts to have $\lfloor u^T A \rfloor = u^T A$, i.e., a rounding of the left hand side is avoided. Caprara and Fischetti [8] propose a number of reduction rules to limit the size of the separation problem.

Contribution. This paper reports on our study to separate general $\{0, \frac{1}{2}\}$-cuts effectively, despite its $\mathcal{NP}$-completeness. We recall that the 0 and $\frac{1}{2}$ coefficients of the vector u allow to reduce the size of $\{0, \frac{1}{2}\}$-SEP considerably by an extended set of preprocessing steps, ranging from obvious observations to a sophisticated procedure based on Gaussian elimination to eliminate rows and columns. After preprocessing, violated $\{0, \frac{1}{2}\}$-cuts can often be indicated directly as single rows of the reduced problem. Our computational experiments show that this is a very vital idea generating many violated $\{0, \frac{1}{2}\}$-cuts with small effort.

Independently from the preprocessing, an ILP is formulated to find the most violated $\{0, \frac{1}{2}\}$-cut. This auxiliary ILP can be solved either for the original separation problem or the reduced one. In a computational study we show that the exact separation can be sped up by a factor of at least 10 if preprocessing is performed first.

The effect of the separation of $\{0, \frac{1}{2}\}$-cuts on the performance of state-of-the-art ILP solvers is documented in a further computational study. It shows that by exact separation the number of branch&cut nodes is reduced by 20% on average at the cost of increased overall computation times due to the auxiliary ILP that has to be solved. Moreover, it is unclear whether the most violated $\{0, \frac{1}{2}\}$-cut is also the one that strengthens the formulation the most. Therefore, we additionally propose a heuristic routine (after preprocessing) to find violated $\{0, \frac{1}{2}\}$-cuts that are likely to strengthen the formulation. Computational experiments show that in such a way the overall computation times can be sped up by 20% for moderately sized instances.

Recently, Fischetti and Lodi [13] followed independently a similar integer programming approach as to optimize over the first Chvátal closure, i.e., the polytope derived by adding all inequalities (2) with $u \in [0, 1)^m$. In contrast to their approach, we can exploit the addressed preprocessing techniques for $\{0, \frac{1}{2}\}$-SEP as to reduce the problem size. By this, we can optimize more effectively over the first Chvátal closure in case the $\{0, \frac{1}{2}\}$-cuts are the only undominated CG cuts, e.g., for the matching polytope.

Outline. The rest of the paper is organized as follow. This section is completed with some further notation used in this paper. Section 2 is dedicated to preprocessing for $\{0, \frac{1}{2}\}$-SEP, whereas exact and heuristic separation algorithms for $\{0, \frac{1}{2}\}$-SEP are presented in Section 3. In Section 4 we report on the results of the computational studies on the effectiveness of the developed ideas and algorithms. The paper is closed with concluding remarks in Section 5.

Notation. Let e^j denote a unit vector of appropriate size with j-th coefficient equal to one, whereas $\mathbf{1}$ ($\mathbf{0}$) denotes the all one (zero) vector and $\mathbb{1}_I(i)$ the indicator function being 1 if $i \in I$ and 0 otherwise. With modulo applied component-wise, we define $\bar{A} = A \mod 2$ and $\bar{b} = b \mod 2$. Moreover, for a fractional solution $x^* \in P_{LP}$, we set $s = b - Ax^* \geq 0$ as slack vector. The violation of (2) for a vector $u \in \{0, \frac{1}{2}\}^m$ and fractional solution $x^* \in P_{LP}$ is denoted by $z(u, x^*) := \lfloor u^T A \rfloor x^* - \lfloor u^T b \rfloor$.

2 Preprocessing $\{0, \frac{1}{2}\}$-SEP

To find a separating $\{0, \frac{1}{2}\}$-cut, we seek for a weight vector u such that $z(u, x^*) > 0$. The next lemma restates this task.

Lemma 1. *Let $x^* \in P_{LP}$ be a fractional solution. There exists a vector $u \in \{0, \frac{1}{2}\}^m$ such that $z(u, x^*) > 0$ if and only if there exists a binary vector $v \in \{0, 1\}^m$ such that $v^T \bar{b}$ is odd and*

$$v^T s + (v^T \bar{A} \mod 2)x^* < 1 \tag{3}$$

holds.

Proof. The violation $z(u, x^*)$ can be rewritten as follows:

$$\begin{aligned} z(u, x^*) &= \lfloor u^T A \rfloor x^* - \lfloor u^T b \rfloor \\ &= \tfrac{1}{2}\left((2u)^T b \mod 2\right) - u^T s - \tfrac{1}{2}\left((2u)^T A \mod 2\right) x^* \\ &\overset{v:=2u}{=} \tfrac{1}{2}\left((v^T \bar{b} \mod 2) - v^T s - (v^T \bar{A} \mod 2)x^*\right) \end{aligned}$$

Since $\bar{A}, \bar{b}, s$, and $v = 2u$ are all non-negative, the only way to obtain a positive violation $z(u, x^*)$ consists in $v^T\bar{b} \mod 2 \equiv 1$ and $v^T s + (v^T\bar{A} \mod 2)x^* < v^T\bar{b} \mod 2 \equiv 1$. □

Note that both conditions in Lemma 1 are independent of the actual values of coefficients and right hand sides, but take into account only their parities, i.e., whether they are even or odd. The vector v indicates the original inequalities to combine with weight $\frac{1}{2}$ such that the right hand side is in fact rounded down, and this strengthening (by $\frac{1}{2}$) is not compensated by the collected slacks together with the necessary rounding of the fractional left hand side coefficients.

In order to simplify the restated task, the system $(\bar{A}, \bar{b}, s)$ and x^* can be preprocessed by a series of transformations and problem size reductions, see also Caprara and Fischetti [8]. The following observations are helpful in this regard:

Lemma 2. *The reductions below do not influence the set of undominated $\{0, \frac{1}{2}\}$-cuts for the original system (and can be assumed to be carried out for the follow ups):*

(i) All columns in $\bar{A}$ corresponding to variables $x_i^* = 0$ can be removed.
(ii) Zero rows in $(\bar{A}, \bar{b})$ can be removed.
(iii) Zero columns in $\bar{A}$ can be removed.
(iv) Identical columns in $\bar{A}$ can be replaced by a single representative with associated variable value as sum of the merged variables.
(v) Any unit vector column $\bar{a}^i = e^j$, $1 \le j \le m$, in $\bar{A}$ can be removed provided that x_i^* is added to the slack s_j of row j.
(vi) Any row $1 \le j \le m$ with slack $s_j \ge 1$ can be removed.
(vii) Rows identical in $(\bar{A}, \bar{b})$ can be eliminated except for one with smallest slack value.

For the proof we refer to [19]. As a result, we obtain a reduced system which is equivalent for the separation. For notational convenience, we continue to use m and n for the (reduced) numbers of rows and columns, respectively. Moreover, we assume throughout the sequel that for any arising interim system, all of these reductions are applied as well.

So far, any row of the system $(\bar{A}, \bar{b}, s)$ represents a single original inequality. A further reduction in problem size can be obtained by row combinations according to rules specified below. For this, we associate with each row j of $(\bar{A}, \bar{b}, s)$ an index set R_j holding the indices of original inequalities currently combined for this row. These index sets are initialized by $R_j = \{j\}$.

We consider a basic operation performed on the rows of $(\bar{A}, \bar{b}, s)$: the addition of one row to another one, where the coefficients of $\bar{A}$ and $\bar{b}$ are added in modulo 2 arithmetic, the coefficients of s in normal arithmetic, and the symmetric difference is taken for the associated index sets. So, adding row i to row j gives a new row j with the following values: $\bar{a}_{jk} := \bar{a}_{ik} + \bar{a}_{jk} \mod 2 \ \forall k$, $\bar{b}_j := \bar{b}_i + \bar{b}_j \mod 2$, $s_j := s_i + s_j$, and $R_j := R_i \triangle R_j$, where $X \triangle Y = (X \cup Y) \setminus (X \cap Y)$ for sets X, Y.

Using this operation, the system $(\bar{A}, \bar{b}, s)$ can be further transformed ad might then allow for additional application of reduction rules from Lemma 2. Except for this, we are particularly interested in rows with zero coefficients and non-zero right hand side.

Lemma 3. *Let j be the index of a zero row in $\bar{A}$ with $\bar{b}_j = 1$. If $s_j < 1$, then the weight vector u defined by $u_i = \frac{1}{2}$ for all $i \in R_j$ and 0 otherwise, defines a violated $\{0, \frac{1}{2}\}$-cut on the original system (A, b, s)*

Proof. Let $v = e^j$. Then $v^T \bar{b} = 1$ and the left hand side of (3) equals $s_j < 1$, and thus by Lemma 1 a violated $\{0, \frac{1}{2}\}$-cut inequality is found. By construction, the index set R_j defines exactly the original inequalities to be combined. □

Notice that adding a row i with slack zero twice to any other row j results in the original row j. Such rows play a key role in the next reduction rule.

Proposition 1. *Let i be the index of a row and k the index of a column of $\bar{A}$ such that $\bar{a}_{ik} = 1$ and $s_i = 0$. Then column k can be removed from $\bar{A}$ provided that row i is added to all other rows j with $\bar{a}_{jk} = 1$ and the slack of row i is set to $s_i := x^*_k$.*

Sketch of Proof. We have to assure that the solutions before and after the removal of column k incur the same violation. Consider the corresponding set of rows before/after a reduction. If row i is contained an odd number of times in the set after reduction, it either is part of the set before reduction and added to an even number of rows, or it is not part of the set before reduction, but added to an odd number of rows of the set before reduction. A case analysis reveals that the same violation is found by setting the slack of row i to x^*_k. The full proof can be found in [19].

In case a zero row in $\bar{A}$ is constructed by (repeated) application of Proposition 1, we either have a row with $\bar{b}_j = 0$ and Lemma 2 (ii) can be applied to remove the row as well, or $\bar{b}_j = 1$, and, by Lemma 3, the row describes a $\{0, \frac{1}{2}\}$-cut with violation $1 - s_j$. If in addition $s_j = 0$, the violation is maximal. By Lemma 1 on the other hand, a $\{0, \frac{1}{2}\}$-cut with maximal violation can be only combined from rows with slack zero and parity sum zero (modulo 2) for all columns k with $x^*_k > 0$. Since the above procedure can be applied as long as there are rows with slack zero, it provides a polynomial-time exact algorithm for maximally violated $\{0, \frac{1}{2}\}$-cuts. Caprara et al. [9] observed the same in the more general context of mod-k-cuts.

Further, a zero row j in $\bar{A}$ with $\bar{b}_j = 1$ and $s_j = 0$ can be very helpful in generating further violated $\{0, \frac{1}{2}\}$-cuts: Any other row i with $\bar{b}_i = 0$ and $s_i < 1$ can be turned into a violated $\{0, \frac{1}{2}\}$-cut by adding row j. This way, a zero row with right hand side 1 is generated, whereas the slack remains the same. Only in case $R_i \cap R_j = \emptyset$, the $\{0, \frac{1}{2}\}$-cut is dominated by the one identified by row j. Therefore if such a row j exists, the condition $u^T b = 1$ can be neglected in the search for further violated cuts.

Corollary 1. *Let i be the index of a row and k the index of a column of $\bar{A}$ such that $\bar{a}_{ik} = 1$, $s_i = 0$, and $x^*_k \geq 1$. Then both row i and column k can be removed from $\bar{A}$ provided that row i is added to all other rows j with $\bar{a}_{jk} = 1$.*

Proof. After application of Proposition 1, $s_i = x^*_k > 1$ and thus Lemma 2 (vi) can be applied to remove row i. □

The above holds in particular for tight upper bound constraints from the original system. If $\bar{b}_i = 0$, only the index sets R_j have to be updated. If $\bar{b}_i = 1$, $\bar{b}_j$ have to be adapted additionally. Corollary 1 further indicates that it is benefical to perform Proposition 1 on

columns with large x^*_k. Altogether, the combination of Lemma 2 and Proposition 1 provides an algorithmic framework for preprocessing the system $(\bar{A}, \bar{b}, s)$ and generation of maximally violated $\{0, \frac{1}{2}\}$-cuts.

3 Separation Algorithms

3.1 Exact Separation

With or without preprocessing, the separation problem $\{0, \frac{1}{2}\}$-SEP can be described by a system $(\bar{A}, \bar{b}, s)$ and a fractional solution $x^* \in P_{LP}$. The exact separation problem can be modeled by an auxiliary integer linear program which maximizes the violation. By Lemma 1, $\{0, \frac{1}{2}\}$-SEP can be restated as the search for a binary weight vector $v \in \{0,1\}^m$ such that $v^T \bar{b} \mod 2 = 1$ and (3) is satisfied. These weights are used as binary variables v_i in the formulation.

Condition (3) requires to determine $u^T \bar{A} \mod 2 \in \{0,1\}^n$. To this end, the variables $y_i \in \{0,1\}$ for all $i = 1, \ldots, n$ are introduced to express whether the i-th variable's coefficient becomes odd in the indicated inequality sum or not. To model the modulo 2 computations, we further need auxiliary integer variables $r = (r_i)_{i=1,\ldots,n} \in \mathbb{Z}^n_+$ for all columns $i = 1, \ldots, n$, as well as an additional $q \in \mathbb{Z}_+$ for the right hand side. The separation problem then reads:

$$\begin{cases} \hat{z} = \min\ s^T v + (x^*)^T y \\ \quad s.t.\ \bar{b}^T v - 2q = 1 \\ \quad\quad \bar{A}^T v - 2r - y = \mathbf{0} \\ \quad\quad v \in \{0,1\}^m \\ \quad\quad y \in \{0,1\}^n \\ \quad\quad r \in \mathbb{Z}^n_+ \\ \quad\quad q \in \mathbb{Z}_+ \end{cases} \tag{4}$$

The optimum value $\hat{z}$ of (4) indicates whether a violated cut has been found or not. If $\hat{z} \geq 1$, this is not the case. If $0 \leq \hat{z} < 1$, $1 - \hat{z}$ equals twice the violation $z(u, x^*)$ of the cut generated by combining the original inequalities which are obtained from symmetric difference by those sets R_j with $v_j = 1$ in the optimum solution.

To guide the search to highly violated $\{0, \frac{1}{2}\}$-cuts, we can add an inequality

$$s^T v + (x^*)^T y \leq 1 - \varepsilon \tag{5}$$

to (4) where $\varepsilon \in (0, 1]$. In this way only cuts with a violation of at least $\frac{1}{2}\varepsilon$ are found.

3.2 Heuristic Search

After the above described reductions, the auxiliary ILP might stay, nevertheless, large for larger ILPs. Hence the search for violated $\{0, \frac{1}{2}\}$-cuts might still be time-consuming. A fast alternative is to enumerate all possible combinations of k or less rows of $(\bar{A}, \bar{b})$ that yield a violated $\{0, \frac{1}{2}\}$-cut with $0 < k \leq m$: First, we check if any single row of $(\bar{A}, \bar{b})$ results in a violated $\{0, \frac{1}{2}\}$-cut. If none of them is violated, we test all combinations of two rows for violation. This process is continued to combinations of k rows,

if all combinations up to $k-1$ rows are not violated or the number of detected violated cuts does not exceed a given limit. Surprisingly, $k = 1$ yielded the best results, cf. Section 4.

4 Computational Results

Framework. We implemented our preprocessing and separation algorithms as additional separator within the branch&cut framework SCIP v0.90 [1,2] using CPLEX 10.01 [16] as underlying LP solver. All of SCIP's standard modules (e.g. separators, heuristics) are kept if not stated differently. SCIP's parameters are set to their default values except for a global time limit of 1 hour per instance and avoidance of restarts during solving.

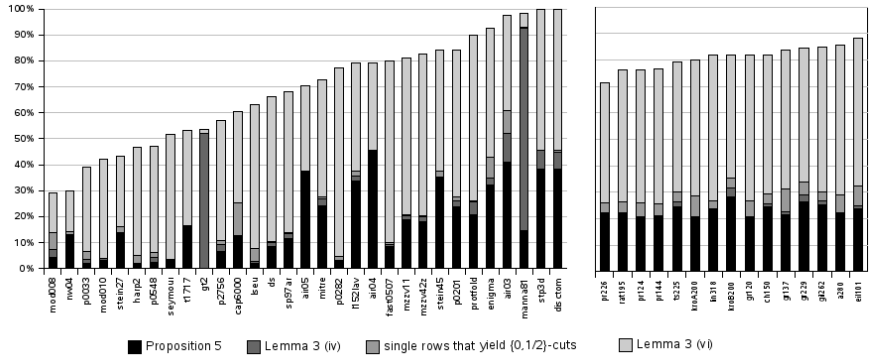
If not stated differently, our separator is called only in the root node like SCIP's standard separators. To investigate the added value of $\{0, \frac{1}{2}\}$-cut separation more accurately our separator is called before SCIP's separators (Gomory, Strong Chvátal-Gomory, Complemented MIR [2]) and cut generating constraint handlers (knapsack, linear). At default, SCIP's separators and constraint handlers are called to separate cuts if and only if our separator does not find a violated cut anymore.

Instead of adding violated $\{0, \frac{1}{2}\}$-cuts directly to the LP, they are stored in a pool from which only the best cuts are selected and added to the LP. We tested several methods to rate the cuts in the pool but we restrict to two methods in the following. The first one is to rate cuts by their violation (i.e., cuts with large violation are better than those with small violation). The second one is similar: Cuts are rated by non-increasing efficacy which is defined as its violation divided by the Euclidian norm of its coefficients (i.e., cuts are better the higher the "average" violation is). The best up to p cuts are transferred to SCIP (with p given as input parameter) which uses further criteria like the parallelism to the objective and other cuts to select the best among all violated inequalities found.

All computations are done on a computer with 3.6 Ghz CPU, 3.7 GB RAM and Linux as operating system. Our computational study includes all pure integer (i.e., non-mixed) instances from MIPLIB 3.0 [6] and MIPLIB 2003 [3] as well as the 2-matching-relaxations of TSP instances from the TSPLIB [5] that also have been studied in [13].

Speed-Up by Preprocessing. We implemented the preprocessing methods suggested in Section 2 in the following order: Removing columns whose corresponding variables (a) are zero in the current LP solution (Lemma 2 (i)) or (b) have a tight variable bound constraint, (c) removing rows with slack at least 1 (Lemma 2 (vi)), (d) removing columns by repeatedly applying Proposition 1, and (e) removing unit vector columns (Lemma 2 (v)). Next we check for empty rows of the preprocessed matrix with a nonzero right hand side (i.e., $\bar{b}_j = 1$). Such a row directly yields a $\{0, \frac{1}{2}\}$-cut. It is (f) deleted from the matrix and if the corresponding cut is violated with violation at least ε, it is added to the pool because every further combination of rows containing such a row cannot yield a stronger cut. Finally, (g) we erase identical rows except for one with the lowest slack value (Lemma 2 (vii)).

Whenever a zero column, a zero row, or a row with slack at least 1 results from a preprocessing step, it is removed from $(\bar{A}, \bar{b})$ immediately (e.g., zero rows that result



(a) MIPLIB instances (b) TSPLIB instances

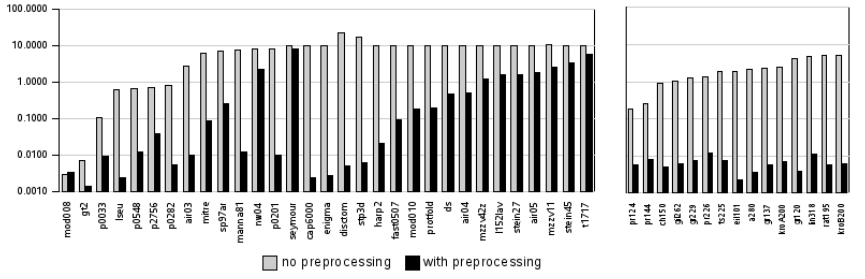
Fig. 1. Efficiency of preprocessing: reduction percentage in number of rows averaged over all applications of the separator

from Proposition 1 are removed from $(\bar{A}, \bar{b})$ and yield a reduction in the number of rows).

To test the effect of preprocessing, we ran all instances with separation of $\{0, \frac{1}{2}\}$-cuts at all nodes of the branch&cut tree without a minimum violation (i.e., $\varepsilon = 0$). The steps (a)-(c) reduce the size of $\bar{A}$ significantly, e.g., considering the MIPLIB instances, on average 83.2% in number of rows (ranging from 46.96% (stein27) to 99.9% (nw04)) and can be applied without greater effort. Considering the 2-matching relaxations of the TSPLIB instances this reduction is even more effective, namely 99.5% of the size of $\bar{A}$ is eliminated (on average). Hence, these steps should certainly be applied and we focus on the further reductions. Figure 1 shows the effect of the steps (d), (e), (f) and (g), using the number of rows as a measure for the problem size. All reduction values are given relative to the number of rows of $\bar{A}$ after applying steps (a)-(c) and are averaged over all times they are applied within the branch&cut. Hence a value of 0% means that no further reduction beside the steps (a)-(c) can be achieved and a value of 100% corresponds to a reduction resulting in an empty pair $(\bar{A}, \bar{b})$. The instances are sorted according to non-decreasing total reduction. Figure 1 shows that on average a reduction of about 14.6% of the remaining size (after applying (a)-(c)) is achieved by applying Proposition 1 (step (d)), 1.26% by removing unit vector columns (step (e)), 1.5% by removing empty rows that yield a violated cut (see above) and finally 40.7% can be achieved by removing identical rows which arise from applying the previous preprocessing methods (in particular within step (d) and (e)).

Moreover the total reduction in number of rows of $(\bar{A}, \bar{b})$ (including the steps (a)-(c)) is increased to 95.5% on average (ranging from 70.0% (stein27) to 99.9% (air03)) for the MIPLIB instances, respectively to 99.9% on average for the 2-matching relaxation of the TSPLIB instances.

This reduction of almost 100% of the size of $\bar{A}$ yields an enormous speed-up in the solving time of the auxiliary ILP (4) as shown in Figure 2. Here the CPU times needed to solve (4) with and without preprocessing (i.e. steps (d) to (g)) are displayed averaged over all auxiliary ILPs within the branch&cut, with a time limit of 10s (i.e., no further



(a) MIPLIB instances (b) TSPLIB instances

Fig. 2. Efficiency of preprocessing: speed-up in solving the auxiliary ILP

nodes of the auxiliary branch&cut are solved as soon as the time limit is exceeded). Note that solving times are given in seconds and we use a logarithmic scale for them. The instances are sorted according non-decreasing average solving time of the auxiliary ILP without preprocessing.

Figure 2 shows that applying the preprocessing steps (d) to (g) reduces the solving time of the auxiliary ILP significantly. Assuming a solving time of 10s for those instances that reach the time limit, for 33% of the MIPLIB instances solving the auxiliary ILP can be sped up by a factor of at least 100. On average over all MIPLIB instances this factor exceeds 38. Considering the 2-matching relaxations, the solving time is sped up by a factor of at least 30 for all instances and exceeds 270 on average. Note that increasing the time limit would yield even higher speed-up factors.

Effect of Separation. To identify the effect of $\{0, \frac{1}{2}\}$-cut separation, two natural key values are available for comparison: the number of nodes of the branch&cut tree and the overall CPU time. Since the 2-matching polytope is completely described by the model inequalities and all $\{0, \frac{1}{2}\}$-cuts (in fact only the blossom inequalities suffice [11]), no branching is needed for these instances if the $\{0, \frac{1}{2}\}$-cuts are separated exactly. Experiments documented in [19] have shown that we are able to solve 80% of the 2-matching relaxations of the TSP instances with less cuts than Fischetti and Lodi [13] need with their more general Chvátal-Gomory cut separation. Here, we consider the pure integer problems from MIBLIB that can be solved within 1 hour with SCIP's default settings. We first compare on the number of branch&cut nodes needed with and without $\{0, \frac{1}{2}\}$-cut separation. For this, we use the following settings: $\{0, \frac{1}{2}\}$-cuts are separated exactly using the auxiliary ILP (4). The separator is called in every node of the branch&cut tree up to a depth of 15. Note that not only violated cuts obtained from the optimal solution of the auxiliary ILP, but also from earlier (non-optimal) solutions are added to the pool. In addition, we apply a simple postprocessing: All single rows whose corresponding variables are zero in the auxiliary ILP solutions (i.e., rows that are not part of the most violated $\{0, \frac{1}{2}\}$-cut yet) are checked. If one of these rows yields a violated $\{0, \frac{1}{2}\}$-cut, it is added to the pool as well. This way, the number of branch&cut nodes needed to solve a problem can be reduced by 26% on average (ranging from a reduction by 84% to an increase by 157%) at the cost of a higher overall solving time: an increase by 158% on

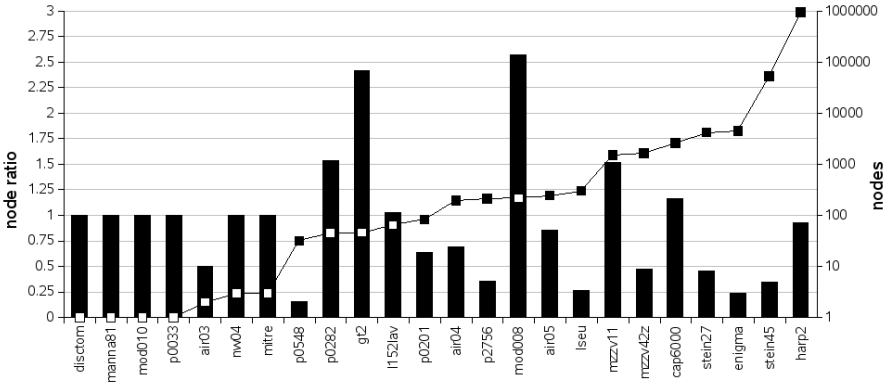


Fig. 3. Efficiency of separation: ratio of branch&cut nodes without and with $\{0, \frac{1}{2}\}$-cuts [bars] and absolute numbers without [line with markers]

average over all instances (primary induced by fast instances with solving times of less than a minute). The results are shown in Figure 3.

Since the computation of an optimal solution to the auxiliary ILP (4) is time consuming and results in few violated cuts , such an approach is not suitable for integration in general purpose ILP solvers. Therefore, we finally perform a CPU time comparison:

(i) SCIP default

(ii) SCIP with our implementation as additional separator using the auxiliary ILP (4) to separate $\{0, \frac{1}{2}\}$-cuts exactly at the root only (cut&branch). Like in the test we used to compare on the branch&cut nodes, not only violated cuts obtained from the optimal solution of the auxiliary ILP, but also from earlier (non-optimal) solutions and from single rows not part of the most violated $\{0, \frac{1}{2}\}$-cut are added to the pool.

(iii) SCIP with our separator using the heuristic described in Section 3.2 to separate $\{0, \frac{1}{2}\}$-cuts at the root node only. Results of (ii) showed us that almost all added $\{0, \frac{1}{2}\}$-cuts are generated from a relative small number of rows of $\bar{A}$: on average only 2 or less "preprocessed" rows. The "original" rows (i.e., rows in A) implied by the preprocessing exceeds 10 on average and goes up to as high as 351 (mzzv11). Inspired by this observation we studied several settings for k. Based on the results of this study we set $k = 1$, i.e., we check all single rows of $(\bar{A}, \bar{b})$ if they yield a violated $\{0, \frac{1}{2}\}$-cut.

In all cases we restrict on those $\{0, \frac{1}{2}\}$-cuts with violation greater than 0.35 (i.e., $\varepsilon = 0.7$ in (5)) to avoid the generation of many weak cuts. Hence not all violated $\{0, \frac{1}{2}\}$-cuts are separated. We add all violated $\{0, \frac{1}{2}\}$-cuts from preprocessing to the pool and additionally up to 100 violated $\{0, \frac{1}{2}\}$-cuts found by the procedures of case (ii) respectively (iii). The $p = 100$ best (w.r.t. their efficacy) cuts of the pool are added to SCIP which decides if they enter the LP (as it does for all its standard separators, as well).

Figure 4 shows the relative solving times of cases (ii) and (iii) w.r.t. case (i), e.g., a value of 0.8 means that solving the instance takes only 80% time compared to SCIP default (case (i)). There are two bars for each instance, the first one refers to case (ii), the

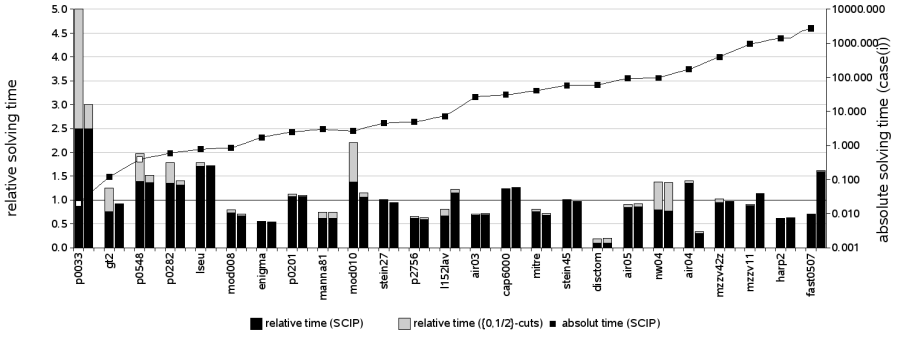


Fig. 4. Efficiency of separation: quotient of solving times of cases (ii) and (iii) w.r.t. case (i) with a split in SCIP and separation time [bars], as well as absolute solving time of case (i) [line with markers]

second one to case (iii). Each bar is divided into two parts: the lower black part shows the fraction of the solving time spent by SCIP methods and the upper grey part shows the fraction spent within the $\{0, \frac{1}{2}\}$-cuts separator. The black boxes refer to the second y-axis on the right hand side which displays the absolute solving times of SCIP default (case (i)) in seconds, according to which the instances are non-decreasingly sorted.

From Figure 4, we conclude that our heuristic approach (case (iii)) performs mostly faster than the exact approach (case (ii)) (with a few exceptions like mzzv11 or fast0507). In particular, the more difficult instances in case (i) with solving times of more than 10 seconds (i.e., instances air03 and on) can often be solved faster. A slowed-down instance like nw04 might be sped up by a faster implementation of our separator, reducing the additional time. Finally we observe that our heuristic approach (case (iii)) reduces the solving time by 7% averaged over all instances (geometric mean), and by 21% averaged over the instances with absolute solving time greater than 10 seconds in case (i).

5 Conclusion

In this paper, we have developed algorithms to separate $\{0, \frac{1}{2}\}$-Chvátal-Gomory cuts in general integer programs, despite the $\mathcal{NP}$-completeness of the problem. Preprocessing rules turned out to be an indispensable part of such algorithms as they do not only reduce the size of the remaining separation problem but in many cases also provide violated inequalities directly, canceling the need for further processing. Separating the most violated $\{0, \frac{1}{2}\}$-cut yields a substantial reduction of the number of branch&cut nodes to be searched until optimality can be proven, whereas heuristic separation achieved the best time performance. The savings in computation time have already aroused the interest of both commercial and academic developers of integer programming solvers.

The developed algorithms are at present only applicable to pure integer programs, i.e., without continuous variables. The extension of the separation procedure to general mixed integer programming problems remains as an important further research direction as these would enhance ILP-solvers further.

References

1. Achterberg, T.: SCIP – a framework to integrate constraint and mixed integer programming. ZIB-Report 04–19, Zuse Institute Berlin (2004), http://www.zib.de/Publications/abstracts/ZR-04-19/
2. Achterberg, T., Berthold, T., Koch, T., Martin, A., Wolter, K.: SCIP (Solving Constraint Integer Programs) (2006), http://scip.zib.de/
3. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003 (2003), http://miplib.zib.de
4. Andreello, G., Caprara, A., Fischetti, M.: Embedding cuts in a branch&cut framework: a computational study with $\{0, \frac{1}{2}\}$-cuts. Technical report, University of Padova (2003)
5. Bixby, R., Reinelt, G.: TSPLIB, http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html
6. Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.P.: MIPLIB 3.0, http://www.caam.rice.edu/~bixby/miplib/miplib.html
7. Caprara, A., Fischetti, M.: $\{0, 1/2\}$-Chvátal-Gomory cuts. Mathematical Programming 74, 221–235 (1996)
8. Caprara, A., Fischetti, M.: Odd cut-sets, odd cycles, and $0 - 1/2$ Chvátal-Gomory cuts. Ricerca Operativa 26, 51–80 (1996)
9. Caprara, A., Fischetti, M., Letchford, A.N.: On the separation of maximally violated mod-k cuts. Mathematical Programming 87, 37–56 (2000)
10. Chvátal, V.: Edmonds polytopes and a hierarchy of combinatorial problems. Discrete Mathematics 4, 305–337 (1973)
11. Edmonds, J., Johnson, E.L.: Matching: a well-solved class of integer linear programs. In: Guy, R.K., Hanani, H., Sauer, N. (eds.) Combinatorial Structures and Their Applications, pp. 80–92. Gordon and Breach, New York (1970)
12. Fiorini, S.: $\{0, \frac{1}{2}\}$-cuts and the linear ordering problem: Surfaces that define facets. SIAM Journal on Discrete Mathematics 20(4), 893–912 (2006)
13. Fischetti, M., Lodi, A.: Optimizing over the first Chvátal closure. Math. Programming 110(1), 3–20 (2007)
14. Gomory, R.E.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society 64, 275–278 (1958)
15. Gomory, R.E.: An algorithm for integer solutions to linear programs. In: Graves, R.L., Wolfe, P. (eds.) Recent Advances in Mathematical Programming, pp. 269–302. McGraw-Hill, New York (1963)
16. ILOG. CPLEX version 10.0 (2006), http://www.ilog.com/products/cplex
17. Koster, A.M.C.A., Zymolka, A.: Stable Multi-Sets. Mathematical Methods of Operations Research 56(1), 45–65 (2002)
18. Koster, A.M.C.A., Zymolka, A.: On cycles and the stable multi-set polytope. Discrete Optimization 2(3), 241–255 (2005)
19. Koster, A.M.C.A., Zymolka, A., Kutschka, M.: Algorithms to separate $\{0, \frac{1}{2}\}$-Chvátal-Gomory cuts. ZIB-Report 07–10, Zuse Institute Berlin (2007), http://www.zib.de/Publications/abstracts/ZR-07-10/
20. Padberg, M.: On the facial structure of set packing polyhedra. Mathematical Programming 5, 199–215 (1973)

Fast Lowest Common Ancestor Computations in Dags

Stefan Eckhardt, Andreas Michael Mühling, and Johannes Nowak

Fakultät für Informatik, Technische Universität München,
Boltzmannstraße 3, D-85748 Garching, Germany
{eckhardt,muehling,nowakj}@in.tum.de

Abstract. This work studies lowest common ancestor computations in directed acyclic graphs. We present fast algorithms for solving the ALL-PAIRS REPRESENTATIVE LCA and ALL-PAIRS ALL LCA problems with expected running time of $O(n^2 \log n)$ and $O(n^3 \log \log n)$ respectively, where the expectation is taken over a distribution of input graphs. The speed-ups over recently developed methods are achieved by applying transitive reduction on the input dags. The algorithms are experimentally evaluated against previous approaches demonstrating a significant improvement. On the purely theoretical side, we improve the upper bound for ALL-PAIRS ALL LCA to $O(n^{3.3399})$. We give first fully dynamic algorithms for both ALL-PAIRS REPRESENTATIVE LCA and ALL-PAIRS ALL LCA. Here, the non-trivial update complexities are $O(n^{2.5})$ and $O(n^3)$ respectively, with constant query times.

1 Introduction

Causality systems or other kinds of entity dependencies are naturally modeled by directed acyclic graphs (dags). Thinking of causal relations among a set of events, natural questions come up, such as: which event entails two given events? What is the last event which entails two given events? Transferring these questions to dags, answers are found by computing common ancestors (CAs), i.e., vertices that reach via some path each of the given vertices, and computing lowest common ancestors (LCAs), i.e., those common ancestors that do not reach any other common ancestor of the two given vertices.

Although LCA algorithms for general dags are indispensable computational primitives, they have been found an independent subject of studies only recently [7, 6, 19]. There is a lot of sophisticated work devoted to LCA computations for the special case of trees (see, e.g., [15, 22, 6]), but due to the limited expressive power of trees they are often applicable only in restrictive or over-simplified settings. There are numerous applications for LCA queries in dags, e.g., object inheritance in programming languages, lattice operations for complex systems, lowest common ancestor queries in phylogenetic networks, or queries concerning customer-provider relationships in the *Internet*. For a more detailed description of possible applications, we refer to [6, 5]. The algorithmic variants studied in this paper are: ALL-PAIRS REPRESENTATIVE LCA : Compute one (representative) LCA for each pair of vertices, and ALL-PAIRS ALL LCA : Compute the set of all LCAs for each pair of vertices.

Related Work. LCA algorithms have been extensively studied in the context of trees with most of the research rooted in [1, 24]. The first asymptotically optimal algorithm

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 705–716, 2007.

for the all-pairs LCA problem in trees, with linear preprocessing time and constant query time, was given in [15]. The same asymptotics was reached using a simpler and parallelizable algorithm in [22]. Recently, a reduction to range minimum queries has been used to obtain a further simplification[6]. More algorithmic variants can be found in, e.g., [8,26,25,9].

In the more general case of dags, a pair of nodes may have more than one LCA, which leads to the distinction of *representative* versus *all* LCA solutions. In early research both versions still coincide by considering dags with each pair having at most one LCA. Extending the work on LCAs in trees, in [20], an algorithm was described with linear preprocessing and constant query time for the LCA problem on arbitrarily directed trees (or, causal polytrees). Another solution was given in [2], where the representative problem in the context of object inheritance lattices was studied. The approach in [2], which is based on poset embeddings into boolean lattices yielded $O(n^3)$ preprocessing and $O(\log n)$ query time on lower semi-lattices.

The representative LCA problem on general dags has been studied recently in [7,6,19,5]. The works rely on fast matrix multiplications (currently the fastest known algorithm needs $O(n^\omega)$ operations, with $\omega < 2.376$ [11]) to achieve $\tilde{O}(n^{\frac{\omega+3}{2}})$ [6][1] and $\tilde{O}(n^{2+\frac{1}{4-\omega}})$ [19] preprocessing time on dags with n nodes and m edges. Recently, in [5] and independently [12], the upper bound of the representative LCA problem has been improved to $O(n^{2+\mu})$ by applying rectangular matrix multiplication [10].[2] For sparse dags, in [19,5], $O(nm)$ algorithms have been presented as well. The authors of [5] study variants of the representative LCA problem, namely (L)CA computations in weighted dags and the ALL-PAIRS ALL LCA problem. For the latter problem, an upper bound of $O(n^{3+\mu})$ is shown, as well as algorithms with scaling properties. In [6], a short experimental study of the performance of ALL-PAIRS REPRESENTATIVE LCA algorithms is presented. It is demonstrated, that on random dags, a suboptimal but simple $O(n^3)$ algorithm based on range minimum queries (RMQ) and ancestor lists performs best.

Results. We summarize the technical contributions of this paper. Most of the proofs are omitted due to space limitations and can be found in the full version of the paper [13].

1. We present fast algorithms for solving the ALL-PAIRS REPRESENTATIVE LCA and ALL-PAIRS ALL LCA problems with expected running time of $O(n^2 \log n)$ and $O(n^3 \log \log n)$ respectively. Here, the expectation is taken over a distribution of all possible input dags, i.e., the algorithms are not randomized. We use the term expected running time in this sense throughout this work. The speed-up over recently developed methods is achieved by applying transitive reduction on the input dags.
2. We improve the recently shown upper bound for ALL-PAIRS ALL LCA from $O(n^{3+\mu})$ to $O(n^{\omega(2,1,1)})$. This result is achieved by applying matrix multiplication to computing boolean witness matrices $W^{(v)}$ for each vertex $v \in V$, where $W^{(v)}[x, y] = 1$ indicates that v is a CA, but not an LCA of (x, y).

[1] Throughout this work, we use $\tilde{O}(f(n))$ for $O(f(n) \cdot \text{polylog}(n))$.

[2] Throughout this work, $\omega(x, y, z)$ is the exponent of the algebraic matrix multiplication of a $n^x \times n^y$ with a $n^y \times n^z$ matrix. Let μ be such that $\omega(1, \mu, 1) = 1 + 2\mu$ is satisfied. The fastest known algorithms for rectangular matrix multiplication imply $\mu < 0.575$ and $\omega(2, 1, 1) < 3.3399$.

3. We give first fully dynamic algorithms for both ALL-PAIRS REPRESENTATIVE LCA and ALL-PAIRS ALL LCA . Here, the non-trivial update complexities are $O(n^{2.5})$ and $O(n^3)$ respectively, with constant query times. The dynamic algorithms are based on a technique which reduces the matrix multiplications in the static solutions to transitive closure computations in corresponding dags.
4. We study LCA computations experimentally. We demonstrate that simplicity and the transitive reduction technique are fundamental to practically efficient algorithms. Furthermore, we evaluate the running time of several ALL-PAIRS REPRESENTATIVE LCA and ALL-PAIRS ALL LCA algorithms in different settings.

2 Fast LCA Algorithms

Let $G = (V, E)$ be a directed graph. Throughout this work we denote by n the number of vertices and by m the number of edges. G is a directed acyclic graph (dag) if and only if G contains no cycles. Let $G_{\text{clo}} = (V, E_{\text{clo}})$ denote the transitive closure of G, i.e., the graph having an edge (u, v) if v is reachable from u over some directed path in G. Similarly, let $G_{\text{red}} = (V, E_{\text{red}})$ be the transitive reduction of G, i.e., the smallest graph G' such that $G_{\text{clo}} = G'_{\text{clo}}$. Throughout this work, let $| E_{\text{red}} | = m_{\text{red}}$.

A dag $G = (V, E)$ imposes a partial ordering on the vertex set. Let N be a bijection from V into $\{1, \ldots, n\}$. N is said to be a *topological ordering* if $N(u) < N(v)$ whenever v is reachable from u in G. We refer to a vertex z which has the maximal topological number $N(z)$ among all vertices in a set as the *rightmost* vertex.

Let $G = (V, E)$ be a dag and $x, y, z \in V$. The vertex z is a *common ancestor* (CA) of x and y if both x and y are reachable from z, i.e., (z, x) and (z, y) are in the transitive closure of G. By $\text{CA}(x, y)$, we denote the set of all CAs of x and y. A vertex z is a *lowest common ancestor* (LCA) of x and y if and only if $z \in \text{CA}(x, y)$ and for each $z' \in V$ with $(z, z') \in E_{\text{clo}}$ we have $z' \notin \text{CA}(x, y)$. $\text{LCA}(x, y)$ denotes the set of all LCAs of x and y.

The algorithms we present in this section are based on the algorithms described in [5]. The following lemma is fundamental to the speed-ups that lead to practically fast algorithms.

Lemma 1. *For all $x, y, z \in V$, it holds that $z = \text{LCA}_G(x, y)$ if and only if $z = \text{LCA}_{G_{\text{red}}}(x, y)$.*

The above lemma implies that LCA computations in a dag G can be restricted to the graph G_{red}. Observe, however, that this does not directly apply to LCA problems involving distances [5]. The restriction to the transitive reduction turns out to be of critical importance when designing practically fast algorithms for ALL-PAIRS REPRESENTATIVE LCA and ALL-PAIRS ALL LCA.

The All-Pairs Representative LCA Problem. In [5], a dynamic programming algorithm which solves ALL-PAIRS REPRESENTATIVE LCA in worst case time $O(nm)$ is given. Applying Lemma 1 yields Algorithm APRLCA.

Theorem 2. *Algorithm APRLCA solves* ALL-PAIRS REPRESENTATIVE LCA *in time* $O(n\, m_{\text{red}})$.

Although the modified algorithm offers no theoretical advantage in the worst case, its superior practical performance can be explained by considering the expected value of the parameter m_{red} on random dags. In this section we consider random dags [4] in the $G_{n,p}$ model, i.e., each possible edge in the graph is chosen independently with equal probability p. The following lemma is due to Simon [23].

Lemma 3. *Let $G = (V, E)$ be a random dag in the $G_{n,p}$ model. Let m_{red} be a random variable denoting the number of edges in the transitive reduction of G. Then*

$$\mathbb{E}[m_{\mathrm{red}}] = O(n \log n)$$

Corollary 4. *Let $G = (V, E)$ be a random dag in the $G_{n,p}$ model. Then, the expected running time of Algorithm APRLCA on G is $O(n^2 \log n)$.*

The All-Pairs All LCA Problem. We improve the algorithms for ALL-PAIRS ALL LCA presented in [5] by applying Lemma 1.

Algorithm APA1 is quite natural. First, it computes the transitive closure of G in $O(nm)$. Then, for every vertex z and every pair (x, y) it determines if z is an LCA of (x, y) in time $O(\text{out-deg}(z))$. If z is a CA of (x, y), but none of its children, then z is an LCA of (x, y). Checking whether a given vertex is a CA of a vertex pair can be done by simple transitive closure look-up. The total running time of the original algorithm is $O(n^2 m)$.

Theorem 5. *The time needed by Algorithm APA1 to solve* ALL-PAIRS ALL LCA *on $G = (V, E)$ is bounded by $O(n^2\, m_{\mathrm{red}})$. Let $G = (V, E)$ be a random dag in the $G_{n,p}$ model. Then, Algorithm APA1 solves* ALL-PAIRS ALL LCA *on G in expected time $O(n^3 \log n)$.*

Applying Lemma 1 to Algorithm 2 in [5] yields Algorithm APA2. Like algorithm APRLCA it is based on dynamic programming. The LCA sets are iteratively constructed by merging already determined LCA sets. In the merging steps, all those vertices that are predecessors of some other vertices in the set are discarded. In [5], it is shown that the merging operation can be performed in time $O(\min\{n, k^2\})$, where k is the maximum cardinality of any LCA set. This finding allows for scaling with the maximal LCA sets. Moreover, it is possible to decide on-line which merging strategy is to be preferred. Observe, however, that even for sparse dags with $m = O(n)$, the total size of the LCA sets can be $\Omega(n^3)$ [5].

Theorem 6. *The time needed by Algorithm APA2 to solve* ALL-PAIRS ALL LCA *on $G = (V, E)$ is bounded by $O(n\, m_{\mathrm{red}} \min\{n, k^2\})$, where k is the maximum LCA set. Let $G = (V, E)$ be a random dag in the $G_{n,p}$ model. Then, Algorithm APA2 solves the* ALL-PAIRS ALL LCA *problem in expected time $O(n^2 \log n \min\{n, k^2\})$.*

Algorithm APA3 is achieved by applying Lemma 1 on Algorithm 4 in [5]. It is based on the following idea. suppose we are given a vertex z and want to determine all pairs (x, y) for which z is an LCA. To this end, we employ an ALL-PAIRS REPRESENTATIVE LCA algorithm on G using a topological ordering that maximizes $N(z)$. Since ALL-PAIRS REPRESENTATIVE LCA algorithms return rightmost LCAs, this approach guarantees

that z is returned for the appropriate pairs. This can even be improved by maximizing the topological numberings of vertices on a path simultaneously. For more details, we refer to [5].

Theorem 7. *The time needed by Algorithm APA3 to solve* ALL-PAIRS ALL LCA *on* $G = (V, E)$ *is bounded by* $O(n\, m_{\mathrm{red}}\, w)$*, where* w *is the size of path cover. Let* $G = (V, E)$ *be a random dag in the* $G_{n,p}$ *model. Then, Algorithm APA3 solves* ALL-PAIRS ALL LCA *in expected time* $O(n^3)$ *for* $\log^2 n/n \leq p < 1$ *and expected running time* $O(n^3 \log\log n)$ *for* $0 < p < \log^2 n/n$.

Proof. The running time of algorithm APA3 is bounded by $O(n\, m_{\mathrm{red}}\, w)$, where w is the width of the path cover of G. To analyze the expected running on a random dag, we use the following lemma [23]:

Lemma 8. *Let* $G = (V, E)$ *be a random dag in the* $G_{n,p}$ *model. Then, there exists an algorithm that computes a path cover of* G *of width* w *and the transitive reduction of* G *in time* $O(w\, m_{\mathrm{red}})$*. Moreover*

$$\mathbb{E}[w\, m_{\mathrm{red}}] = \begin{cases} O(n^2) & \text{for } \log^2 n/n \leq p < 1 \\ O(n^2 \log\log n) & \text{otherwise} \end{cases}$$

Algorithm B in [23] with running time $O(n+m)$ yields a path cover of size w satisfying the conditions of the above lemma. Hence, the theorem follows. □

3 Theoretical Improvements

Improved Upper Bounds for ALL-PAIRS ALL LCA. We start by considering the following problem which we call ONE-VERTEX ALL-PAIRS LCA . Given a dag G and a vertex v, find all pairs (x, y) such that v is an LCA of (x, y). Clearly, from a ONE-VERTEX ALL-PAIRS LCA solution for each $v \in V$, a solution to ALL-PAIRS ALL LCA can be derived in $O(n^3)$.

ONE-VERTEX ALL-PAIRS LCA reduces to the following. Find all pairs (x', y') such that v is a CA of (x', y'). Then, test for each such pair, if there exists a witness, i.e., a successor of v that is also a CA of this pair, or not. To this end, it is enough to consider the children of v in G.

A solution to this problem works as follows. We initialize an $n \times n$ matrix $C^{(v)}$, such that $C^{(v)}[x, y] = 1$ if and only if v is a CA of (x, y). Obviously, this can be done in time $O(n^2)$ with knowledge of the transitive closure. Let $A^{(v)}$ be an $n \times n$ matrix such that $A^{(v)}[x, z] = 1$ if and only if x is reachable from z and z is a direct child of v, i.e., $(v, z) \in E$. Let $A^{(v)}[x, z] = 0$ otherwise. This can be thought of as the transpose of the transitive closure restricted to the rows indexed with children of v, all other entries set to zero. Let A be the adjacency matrix corresponding to G_{clo}. Let $W^{(v)} = A^{(v)} \cdot A$ be the *witness matrix* of v.

Lemma 9. *Let* $W^{(v)}$ *be the witness matrix of vertex* v*. Then,* v *is an LCA of* (x, y) *if and only if* $W^{(v)}[x, y] = 0$ *and* $C^{(v)}[x, y] = 1$.

Hence, ONE-VERTEX ALL-PAIRS LCA can be solved in time $O(n^\omega)$. This approach leads immediately to an $O(n^{1+\omega})$ algorithm for ALL-PAIRS ALL LCA . Fast rectangular matrix multiplication yields even a stronger upper bound. We can compute the

witness matrices $W^{(v)}$ for vertices $v \in V$ in one step by multiplying an $n^2 \times n$ and an $n \times n$ matrix in time $O(n^{\omega(2,1,1)})$.

$$\begin{pmatrix} A^{(v_1)} \\ A^{(v_2)} \\ \vdots \\ A^{(v_n)} \end{pmatrix} \cdot A = \begin{pmatrix} W^{(v_1)} \\ W^{(v_2)} \\ \vdots \\ W^{(v_n)} \end{pmatrix} \tag{1}$$

Theorem 10. ALL-PAIRS ALL LCA *can be solved in time* $O(n^{\omega(2,1,1)})$.

Proof. The algorithm works as follows:

1. Compute the transitive closure of G
2. Compute the common ancestor matrices $C^{(v)}$ for all $v \in V$. Since G_{clo} is known, this can be achieved in time $O(n^3)$.
3. Compute the witness matrices $W^{(v)}$ for all $v \in V$ using Equation (1). This takes time $O(n^{\omega(2,1,1)})$. Currently, upper bounds for rectangular matrix multiplication imply $\omega(2,1,1) < 3.3399$.
4. Let $L^{(v)} = C^{(v)} - W^{(v)}$ for each $v \in V$. Then, for each pair (x, y), all LCAs can be read from the entries $L^{(v)}[x, y]$ for each v. This step takes a total of $O(n^3)$ time. □

Dynamic Algorithms for ALL-PAIRS REPRESENTATIVE LCA. We show how to solve the fully dynamic ALL-PAIRS REPRESENTATIVE LCA problem with update time $O(n^{2.5})$ and query time $O(1)$. We consider *vertex-centered* updates, that is, given a vertex v, edges incident to v can be arbitrarily added or deleted in one update step. The approach is based on the matrix multiplication-based static ALL-PAIRS REPRESENTATIVE LCA solution in [5]. The static algorithm is based on rectangular matrix multiplications which serve as *CA existence* computations. To this end, the vertices are divided into $l = n^{1-r}$ sets $V^1, \ldots, V^l$ of size n^r for some $r \in [0, 1]$. To ease exposition, we assume in the sequel that n^r and n^{1-r} are integers. For each vertex set V^i one rectangular matrix product of an $n \times n^r$ and an $n^r \times n$ matrix is computed in order to identify all pairs for which a CA in the set V^i exists. The rightmost LCAs are then searched in the vertex set with the largest index i. In order to improve the upper bound for vertex-centered updates, we reduce the rectangular matrix multiplications to transitive closure computations in dags $G^1, \ldots, G^l$. The reduction has the following properties:

1. The adjacency matrices of the dags G^i_{clo} correspond to the results of the rectangular matrix products in the static solution.
2. Vertex-centered updates incur at most a constant amount of vertex-centered updates in each of the dags G^i.

Each of the transitive closures of the dags G^i can be updated in time $O(n^2)$ using recent results [21].

Theorem 11. *There exists an algorithm for dynamic* ALL-PAIRS REPRESENTATIVE LCA *with* $O(n^{1-r+\omega} + n^{2+r})$ *initialization,* $O(n^{1-r+2} + n^{2+r})$ *update, and* $O(1)$ *query time.*

Optimizing r for updates, we get $O(n^{2.876})$ initialization, $O(n^{2.5})$ update, and $O(1)$ query time.

Dynamic Algorithms for ALL-PAIRS ALL LCA. For ALL-PAIRS ALL LCA we achieve $O(n^{1+\omega})$ initialization, $O(n^3)$ update and $O(1)$ query time for fully dynamic vertex-centered updates. Here, the key is to speed up witness computations in the dynamic setting. Again, the dynamic speed-up is achieved by using transitive closure computations instead of matrix products for the witness matrix computations. The application of the reduction technique described above to ALL-PAIRS ALL LCA is straightforward.

Theorem 12. *There exists an algorithm for dynamic* ALL-PAIRS REPRESENTATIVE LCA *with* $O(n^{3.376})$ *initialization,* $O(n^3)$ *update, and* $O(1)$ *query time.*

Observe that $O(n^3)$ update time is optimal in the worst case, since a single update can change up to $\Theta(n^3)$ entries.

4 Experiments

Implementation.[3] We have implemented Algorithms APRLCA, APA1, APA2, and APA3 described in Section 2 in C++. For comparison with two of the algorithms that were subject to the experimental study in [6], we adapted the code provided by the authors to fit in our C++ framework. The third algorithm tested in [6] was ruled out due to its inferior performance. This finding is in accordance with the results of the study presented in [6]. *RMQuery* is based on combining ancestor lists for each of the vertices with LCA queries on a spanning tree of G. The running time of the algorithm is $O(n^3)$, but as a result of the experimental study in [6], it is efficient in practice. *TCQuery* is based on transitive closure queries. It computes G_{clo} first and then chooses the rightmost CA of a pair (x, y) by comparing the corresponding rows of the adjacency matrix of G_{clo}. The running time of the algorithm is $\Theta(n^3)$.

Additionally, we implemented the two transitive closure algorithms described in [23]: the algorithm of Goralćíková and Koubek[14](GK) with expected running time of $O(n^2 \log n)$ and the algorithm of Simon [23] (Simon) with expected running time of $O(n^2 \log \log n)$. Our preprocessing routines are complemented by the greedy algorithm for computing a path cover of G with running time $O(n + m)$[23]. Both of the transitive closure algorithms naturally compute the transitive reduction and are used to accomplish both tasks. Approaches based on fast algebraic matrix multiplication were excluded from this study. These methods are widely believed to be not efficient in practice. The tests were done on a system with an Athlon XP 3000+ clocked at 2.154 GHz with 2 GB RAM and running on Linux.

Test Data. We tested the algorithms on three families of dags:

- $G_{n,p}$ **random dags:** In order to mirror random dags of varying density we use the parameter p to control the expected number of edges in the dag. In all of our experiments we considered sparse ($\mathbb{E}[m] = \Theta(n)$), medium ($\mathbb{E}[m] = \Theta(n \log n)$), and dense $\left(\mathbb{E}[m] = \Theta(n^2)\right)$ graphs.
- **Power law random dags:** Modeling real world data often leads to graphs in which the degrees of the vertices satisfy the power law. That is, the number of vertices of

[3] Our implementations and the used test data is available at http://wwwmayr.informatik.tu-muenchen.de/personen/nowakj/lca/

degree k is proportional to $k^{-\alpha}$ for some constant $\alpha > 1$. This is also true for dags, e.g., a citation graph compiled from crawling scientific literature obey a power law with $\alpha \approx 1.7$ [3]. There are numerous examples of large-scale networks which fall into the category of power law graphs. We generated random dags in which the out-degrees of the vertices have expected degrees following a power law slightly adapting the Chung-Law model [3]. First, target out-degrees of the vertices are generated according to the power law. Suppose that the vertices are sorted in decreasing order according to their target degrees. Then, for each vertex pair (i, j), $i < j$, an edge is added with probability $\deg_i/(n - i)$, where $\deg_i$ is the target degree of i. We generated dags for $\alpha = 2$ and $\alpha = 1.5$, representing common exponents.

– **Real-world data sets:** Our real-world data sets include the *Internet dag*, the *citation dag*, and phylogenetic networks. The Internet dag is derived from business relationships of autonomous systems (ASes) in the Internet. Our data set is obtained from observable BGP routes. From these routes an acyclic AS graph is inferred with the method proposed in [18]. We use the same data set as in [16], where a detailed description of the data collection and postprocessing steps can be found. The final dag has 22,218 vertices and 57,413 edges. The citation dag is compiled from Citeseer's OAI publicly available records[4]. Similar graphs have been studied and analyzed extensively in the past, see [3] and references therein. Computing LCAs in such dags may provide valuable information, e.g., which papers are original to two or more papers on a particular topic. The dag has 716,772 vertices and 1,331,948 edges. The data provided by CiteSeer is automatically obtained by crawling the web and not free of errors. We first parsed the documents and created a citation graph, where an edge (x, y) was added whenever y references x. We then manually cleaned the data by ordering the vertices according to their timestamps (which are part of the data) and deleting edges that are not consistent with the partial order imposed by the timestamps. We considered two phylogenetic networks for which the data sets are included as examples in the SplitsTree4 [17] package. The first network represents evolutionary relationships of a set of mammals. It has 498 vertices and 889 edges. The second network represents evolutionary relationships between a single protein, namely myosin, a protein involved in muscle contraction, in different organisms. The network has 5462 vertices and 10,321 edges.

  The Internet dag and the citation dag had to be sampled in order to be processed by our algorithms. We were interested in dense subgraphs in order to evaluate the effect of the transitive reduction in real-world dags. To achieve this, we sampled the graphs by performing breadth first searches starting at high-degree vertices. To sample a graph of size n, we choose the dag which is induced by the first n visited vertices. The sampling imposes a bias towards dense subgraphs. However, most of the instances generated are still *s*parse in the sense that the average degree of the vertices is a small constant, e.g., less than 5 in the case of the citation graph samples.

  Although the effect of the transitive reduction on sparse dags is small, our $O(nm)$ dynamic programming approach is naturally by far the best choice for such dags.

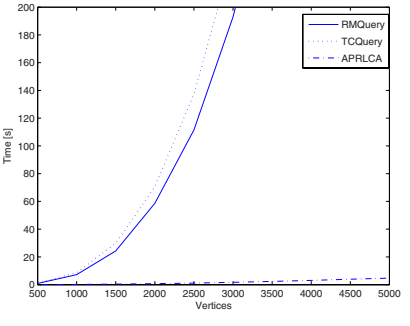
[4] http://citeseer.ist.psu.edu/oai.html

Results. Due to space limitations, we cannot present all experimental data in this work. The full set of results is given in [13].

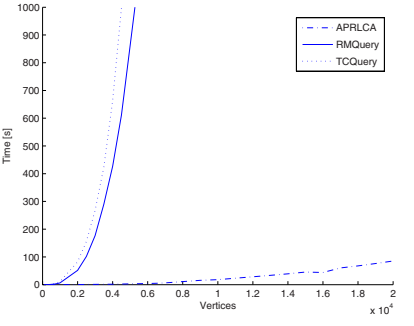
Transitive Reduction: We first compared the two methods for creating the transitive reduction of a dag. Simon clearly outperforms GK on medium and dense dags. Moreover, the space requirement for GK is significantly larger than for Simon imposing a limiting factor on large problem instances. Consequently, this method was used to create the transitive reductions in all further experiments.

We also compared the effect of the transitive reduction in our real world data sets with comparable $G_{n,p}$ dags, i.e., dags having the same number of vertices and same expected number of edges. Interestingly, the effect seems to be larger in the real world data sets. This may imply that—at least in sparse dags—the benefit of using transitive reduction as an algorithmic speed-up is even greater in non-random dags.

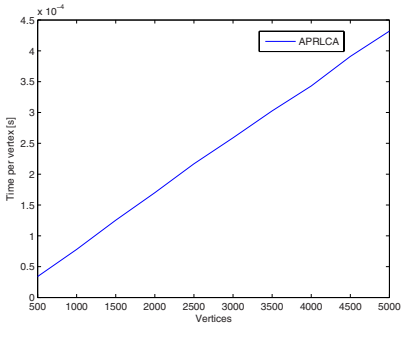
All-Pairs Representative LCA: We compared the performance of Algorithm APRLCA with the two methods presented in [6]. Separately, we tested the result of using transitive reduction with the RMQuery algorithm. The effect is very small compared to APRLCA, but improves the running time slightly. Subsequently, transitive reduction



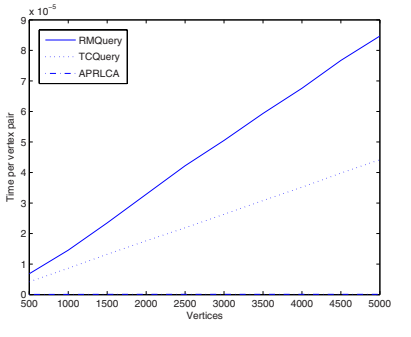
(a) Performance of APR-algorithms on power law dags with $\alpha = 2$



(b) Performance of APR-algorithms on the Internet dag samples



(c) APRLCA as *time per vertex* on dense dags



(d) TCQuery and RMQuery measured as *time per vertex pair* on dense dags

Fig. 1. Evaluation of APR-algorithms

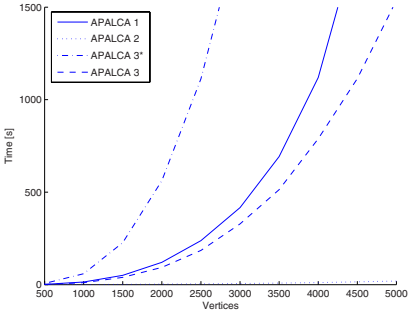
Table 1. Experimental results on phylogenetic networks

	n	m	m_{red}	t(TR)	avg#LCAs	t(RMQuery)	t(TCQuery)	t(APRLCA)	t(APA1)	t(APA2)	t(APA3*)	t(APA3)
Mammals	498	889	847	0.01	1.52574	0.88	1.17	0.02	1.38	1.34	8.7	1.741
Myosin	5462	10321	9997	1.09	1.14211	1159.96	545.623	4.56	1142.27	589.466	1976.1	1540.261

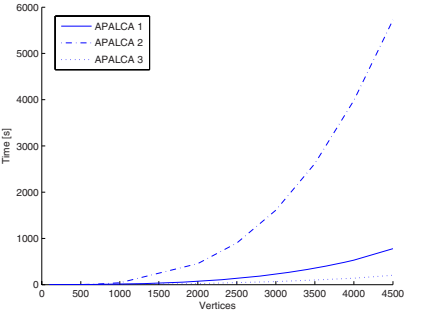
is used for APRLCA and RMQuery, whereas, TCQuery clearly does not benefit from transitive reduction.

A sample of the results can be seen in Figure 1. The dynamic programming algorithm with transitive reduction proves to be the algorithm of choice. The benefits of using the transitive reduction increase significantly at higher densities. On the other hand, APRLCA with running time $O(nm)$ is naturally the best choice on sparse dags clearly outperforming RMQuery and TCQuery. Experimental results on all families of dags support this finding. The full set of experimental data can be found in [13].

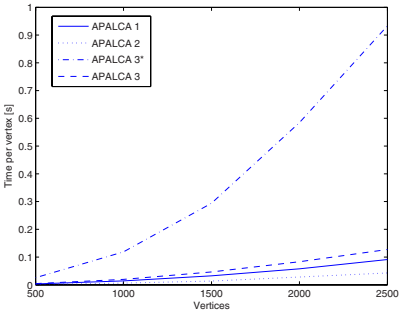
We measured the running times of APRLCA, RMQuery, and TCQuery, as *time per vertex*, i.e., dividing the running times by n, and again RMQuery and TCQuery as *time per vertex pair*, i.e., dividing the times by $n(n-1)/2$. As a result, we conclude that the asymptotic improvement of APRLCA over the two other methods is roughly a factor of n. In particular, Figure 1(d) indicates that both algorithms have a cubic running time.



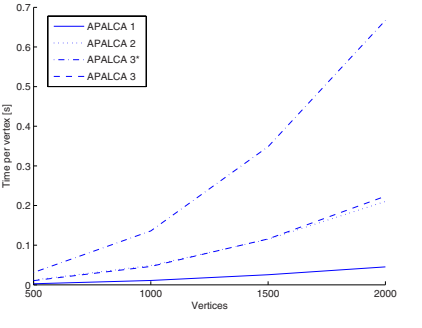
(a) Performance of APA-algorithms on dense $G_{n,p}$ dags



(b) Performance of APA-algorithms on the Internet dag samples



(c) APA-algorithms on medium $G_{n,p}$ dags as *time per vertex*



(d) APA-algorithms on power law dags ($\alpha = 1.5$) as *time per vertex*

Fig. 2. Evaluation of APA-algorithms

The effect of the transitive reduction on the very sparse phylogenetic networks is very small, see Table 1. For ALL-PAIRS REPRESENTATIVE LCA , the number of edges should be roughly halved by the transitive reduction in order to justify the additional expenses for computing the transitive reduction.

All-Pairs All LCA: In addition to APA1, APA2, and APA3, we considered the performance of algorithm APA3*, which is essentially APA3 without using a path cover. The comparison of APA3 and APA3* enables direct evaluation of the effect of using a path cover. The large output complexity of the problem limits the tests to considerably smaller graphs, i.e., $n \leq 5000$. Moreover, on hard problem instances, i.e., instances having large LCA sets, the restriction on the number of vertices is even larger.

A sample of the results is given in Figure 2. APA2 is the best choice on $G_{n,p}$ dags, whereas APA1 leads on power law dags and the real world data sets. Although APA3 has the best guaranteed expected running time, the overhead caused by the more complicated data structures does not pay off for the considered problem sizes. The good performance of APA2 can be explained by the fact that the LCA sets are usually small. The average number of LCAs for a pair is less then 2 in most of our experiments, especially on large, dense dags. Our results suggest that in most cases we can bound the number of LCAs for each pair by a constant. This implies an $O(n^2 \log\log n)$ running time of APA2. This is also supported by plotting the running times of the APA-algorithms per vertex. The effects of using a path cover to maximize the topological number of several vertices in one step over maximizing only a single vertex can be estimated by comparing APA3 and APA3*. While on sparse graphs, the additional cost of computing and maintaining the cover does not pay off, the effects on medium and dense dags are considerable.

Interestingly, medium sized dags turn out to be the most difficult problem instances both in terms of running time and output size. We evaluated the the maximal and average size of the LCA sets in dependence of the dag densities. We conjecture that the values peak out in $G_{n,p}$ dags for $p \approx \text{polylog}(n)/n$ which is supported by our experimental findings. In power law dags, the values roughly peak out at $\alpha = 1.5$. This may explain the relatively inferior performance of APA2 on these dags.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: On finding lowest common ancestors in trees. SIAM J. Comput. 5(1), 115–132 (1976)
2. Aït-Kaci, H., Boyer, R.S., Lincoln, P., Nasr, R.: Efficient implementation of lattice operations. ACM Trans. Prog. Lang. Syst. 11(1), 115–146 (1989)
3. An, Y., Janssen, J., Milios, E.E.: Characterizing and mining the citation graph of the computer science literature. Knowl. Inf. Syst. 6(6), 664–678 (2004)
4. Barak, A.B., Erdős, P.: On the maximal number of strongly independent vertices in a random directed acyclic graph. SIAM J. on Algebraic and Discrete Methods 5(4), 508–514 (1984)
5. Baumgart, M., Eckhardt, S., Griebsch, J., Kosub, S., Nowak, J.: All-pairs common-ancestor problems in weighted directed acyclic graphs. In: Proc. ESCAPE'07. LNCS, vol. 4614, pp. 282–293. Springer, Heidelberg (2007)
6. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms 57(2), 75–94 (2005)

7. Bender, M.A., Pemmasani, G., Skiena, S., Sumazin, P.: Finding least common ancestors in directed acyclic graphs. In: Proc. SODA'01, pp. 845–854 (2001)
8. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. Syst. Sci. 48(2), 214–230 (1994)
9. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. 34(4), 894–923 (2005)
10. Coppersmith, D.: Rectangular matrix multiplication revisited. J. Complexity 13(1), 42–49 (1997)
11. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation 9(3), 251–280 (1990)
12. Czumaj, A., Kowaluk, M., Lingas, A.: Faster algorithms for finding lowest common ancestors in directed acyclic graphs. ECCC (111) (2006)
13. Eckhardt, S., Mühling, A.M., Nowak, J.: Fast lca computations in directed acyclic graphs. Technical Report TUM-I0707, Inst. f. Informatik, TU München (2006)
14. Goralćíková, A., Koubek, V.: A reduct-and-closure algorithm for graphs. In: Becvar, J. (ed.) Mathematical Foundations of Computer Science 1979. LNCS, vol. 74, pp. 301–307. Springer, Heidelberg (1979)
15. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
16. Hummel, B., Kosub, S.: Acyclic type-of-relationship problems on the internet: An experimental analysis. Technical report, Inst. f. Informatik, TU München (2007)
17. Huson, D.H., Bryant, D.: Application of phylogenetic networks in evolutionary studies. Mol. Biol. Evol. 23(2), 254–267 (2006)
18. Kosub, S., Maaß, M.G., Täubig, H.: Acyclic type-of-relationship problems on the internet. In: Erlebach, T. (ed.) CAAN 2006. LNCS, vol. 4235, pp. 98–111. Springer, Heidelberg (2006)
19. Kowaluk, M., Lingas, A.: LCA queries in directed acyclic graphs. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 241–248. Springer, Heidelberg (2005)
20. Nykänen, M., Ukkonen, E.: Finding lowest common ancestors in arbitrarily directed trees. Inf. Process. Lett. 50(1), 307–310 (1994)
21. Sankowski, P.: Dynamic transitive closure via dynamic matrix inverse (extended abstract). In: Proc. FOCS'04, pp. 509–517 (2004)
22. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. 17(6), 1253–1262 (1988)
23. Simon, K.: An improved algorithm for transitive closure on acyclic digraphs. Theor. Comput. Sci. 58, 325–346 (1988)
24. Tarjan, R.E.: Applications of path compression on balanced trees. J. ACM 26(4), 690–715 (1979)
25. Wang, B.-F., Tsai, J.-N., Chuang, Y.-C.: The lowest common ancestor problem on a tree with an unfixed root. Information Sciences 119(1–2), 125–130 (1999)
26. Wen, Z.: New algorithms for the LCA problem and the binary tree reconstruction problem. Inf. Process. Lett. 51(1), 11–16 (1994)

A Practical Efficient Fptas for the 0-1 Multi-objective Knapsack Problem

Cristina Bazgan, Hadrien Hugot, and Daniel Vanderpooten

LAMSADE, Université Paris Dauphine, Place Du Maréchal De Lattre de Tassigny, 75 775 Paris Cedex 16, France
{bazgan,hugot,vdp}@lamsade.dauphine.fr

Abstract. In the present work, we are interested in the practical behavior of a new fptas to solve the approximation version of the 0-1 multi-objective knapsack problem. Nevertheless, our methodology focuses on very general techniques (such as dominance relations in dynamic programming) and thus may be applicable in the implementation of fptas for other problems as well. Extensive numerical experiments on various types of instances establish that our method performs very well both in terms of CPU time and size of solved instances. We point out some reasons for the good practical performance of our algorithm. A comparison with an exact method is also performed.

Keywords: Multi-objective knapsack problem, approximation, dynamic programming, dominance relations, combinatorial optimization.

1 Introduction

In multi-objective combinatorial optimization, a major challenge is to generate either the set of efficient solutions, that have the property that no improvement on any objective is possible without sacrificing on at least another objective, or the set of non-dominated criterion vectors corresponding to their image in the criterion space. The reader can refer to [1] about multi-objective combinatorial optimization. However, even for moderately-sized problems, it is usually computationally prohibitive to identify the efficient set for two major reasons. First, the number of efficient solutions can be very large. This occurs notably when solving *intractable* instances of combinatorial multi-objective problems, for which the number of efficient solutions is not polynomial in the size of these instances (see, e.g., [1] about the intractability of multi-objective problems). Second, for most multi-objective problems, deciding whether a given solution is dominated is NP-hard, even if the underlying single-objective problem can be solved in a polynomial time.

To handle these two difficulties, researchers have been interested in developing approximation algorithms with provable guarantee such as fully polynomial approximation schemes (fptas). Indeed, an fptas guarantees to compute, for a given accuracy $\varepsilon > 0$, in a running time that is polynomial both in the size of the input and in $1/\varepsilon$, an $(1+\varepsilon)$-approximation, that is a subset of solutions such that, for each efficient solution, this subset contains a solution that is at most

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 717–728, 2007.

at a factor $(1+\varepsilon)$ on all objective values. This is made possible since it has been pointed out in [2], that, under certain general assumptions, there always exists an $(1+\varepsilon)$-approximation, with any given accuracy $\varepsilon > 0$, whose size is polynomial both in the size of the instance and in $1/\varepsilon$. Thus using an fptas for solving a multi-objective problem has two main advantages: on the one hand it provides us with an efficient algorithm to compute an approximation with a guaranteed accuracy and on the other hand it computes an approximation of reasonable size. Nevertheless, in this stream, researchers are usually motivated by the theoretical question of proving or disproving the existence of an fptas for a given problem. Thus, practical implementations of fptas are cruelly lacking and most of the schemes proposed in the literature are not effective in practice.

We consider in this paper the 0-1 multi-objective knapsack problem which has been shown to admit an fptas in [3,4,5]. Our perspective, however, is to propose another fptas focusing on its practical behavior. The main idea of our approach, based on dynamic programming, relies on the use of several complementary dominance relations to discard partial solutions. In a previous work [6], such techniques have been proved to be extremely effective to solve the exact version of this problem. Extensive numerical experiments on various types of instances are reported and establish that our method performs very well both in terms of CPU time and size of solved instances (up to 20000 items in less than 1 hour in the bi-objective case). We compare our approach with the exact method of [6], which is the most effective exact method currently known. In our experiments, we point out some reasons for the good practical performance of our algorithm that may be applicable to other fptas. Indeed, since our methodology focuses on very general techniques (such as dominance relations in dynamic programming), it may be applicable in the implementation of fptas for other problems as well.

This paper is organized as follows. In section 2, we review basic concepts about multi-objective optimization and approximation, and formally define the 0-1 multi-objective knapsack problem. Section 3 presents the dynamic programming approach using dominance relations. Section 4 is devoted to the presentation of the dominance relations. Computational experiments and results are described in section 5. Conclusions are provided in a final section.

2 Preliminaries

We first recall that, given $\succsim$ a binary relation defined on a finite set A, $B \subseteq A$ is a *covering (or dominating) set* of A with respect to $\succsim$ if and only if for all $a \in A \backslash B$ there exists $b \in B$ such that $b \succsim a$, and $B \subseteq A$ is an *independent (or stable) set* with respect to $\succsim$ if and only if for all $b, b' \in B,\ b \neq b',\ \text{not}(b \succsim b')$.

2.1 Multi-objective Optimization and Approximation

Consider a multi-objective optimization problem with p criteria or objectives where X denotes the finite set of feasible solutions. Each solution $x \in X$ is represented in the criterion space by its corresponding criterion vector $f(x) = (f_1(x), \ldots, f_p(x))$. We assume that each criterion has to be maximized.

From these p criteria, the dominance relation defined on X, denoted by $\underline{\Delta}$, states that a feasible solution x dominates a feasible solution x', $x\underline{\Delta}x'$, if and only if $f_i(x) \geq f_i(x')$ for $i = 1, \ldots, p$. We denote by Δ the asymmetric part of $\underline{\Delta}$. A solution x is *efficient* if and only if there is no other feasible solution $x' \in X$ such that $x'\Delta\ x$, and its corresponding criterion vector is said to be *non-dominated*. The set of non-dominated criterion vectors is denoted by ND. A set of efficient solutions is said to be *reduced* if it contains only one solution corresponding to each non-dominated criterion vector. Observe that $X' \subseteq X$ is a reduced efficient set if and only if it is a covering and independent set with respect to $\underline{\Delta}$.

For any constant $\varepsilon \geq 0$, the relation $\underline{\Delta}_\varepsilon$, called ε-dominance, defined on X, states that for all $x, x' \in X$, $x\underline{\Delta}_\varepsilon x'$ if and only if $f_i(x)(1+\varepsilon) \geq f_i(x')$ for $i = 1, \ldots, p$. For any constant $\varepsilon \geq 0$, an $(1+\varepsilon)$-*approximation* is a covering set of X with respect to $\underline{\Delta}_\varepsilon$. Any $(1+\varepsilon)$-approximation which does not contain solutions that dominate each other, *i.e.* which is independent with respect to $\underline{\Delta}$, is *a reduced* $(1+\varepsilon)$*-approximation*. In the following, for a given reduced $(1+\varepsilon)$-approximation, ND_ε denotes the image in the criterion space of this reduced $(1+\varepsilon)$-approximation.

2.2 The 0-1 Multi-objective Knapsack Problem

An instance of the 0-1 multi-objective knapsack problem consists of an integer capacity $W > 0$ and n items. Each item k has a positive integer weight w^k and p non negative integer profits $v_1^k, \ldots, v_p^k$ $(k = 1, \ldots, n)$. A feasible solution is represented by a vector $x = (x_1, \ldots, x_n)$ of binary decision variables x_k, such that $x_k = 1$ if item k is included in the solution and 0 otherwise, which satisfies the capacity constraint $\sum_{k=1}^{n} w^k x_k \leq W$. The value of a feasible solution $x \in X$ on the ith objective is $f_i(x) = \sum_{k=1}^{n} v_i^k x_k$ $(i = 1, \ldots, p)$. For any instance of this problem, we consider two versions: *the exact version* which aims at determining a reduced efficient set, and *the approximation version* which aims at determining a reduced $(1+\varepsilon)$-approximation.

3 Dynamic Programming for the Approximation Version

We first describe the sequential process used in Dynamic Programming (DP) and introduce some basic concepts of DP (section 3.1). Then, we present the concept of dominance relations for solving the approximation version by a DP approach (section 3.2).

3.1 Sequential Process and Basic Concepts of DP

The sequential process used in DP consists of n phases. At any phase k we generate the set of states S^k which represents all the feasible solutions made up of items belonging exclusively to the k first items $(k = 1, \ldots, n)$. A state $s^k = (s_1^k, \ldots, s_p^k, s_{p+1}^k) \in S^k$ represents a feasible solution of value s_i^k on the ith objective $(i = 1, \ldots, p)$ and of weight s_{p+1}^k. Thus, we have $S^k = S^{k-1} \cup \{(s_1^{k-1} + v_1^k, \ldots, s_p^{k-1} + v_p^k, s_{p+1}^{k-1} + w^k) : s_{p+1}^{k-1} + w^k \leq W, s^{k-1} \in S^{k-1}\}$ for $k =$

$1, \ldots, n$ where the initial set of states S^0 contains only the state $s^0 = (0, \ldots, 0)$ corresponding to the empty knapsack. In the following, we identify a state and a corresponding feasible solution. Thus, relations defined on X are also valid on S^k, and we have $s^k \underline{\Delta} \tilde{s}^k$ if and only if $s_i^k \geq \tilde{s}_i^k$, $i = 1, \ldots, p$ and $s^k \underline{\Delta}_\varepsilon \tilde{s}^k$ if and only if $s_i^k(1+\varepsilon) \geq \tilde{s}_i^k$, $i = 1, \ldots, p$.

Definition 1 (Completion, extension, restriction). *For any state $s^k \in S^k$ $(k \leq n)$, a* completion *of s^k is any, possibly empty, subset $J \subseteq \{k+1, \ldots, n\}$ such that $s_{p+1}^k + \sum_{j \in J} w^j \leq W$. We assume that any state $s^n \in S^n$ admits the empty set as unique completion. A state $s^n \in S^n$ is an* extension *of $s^k \in S^k$ $(k \leq n)$ if and only if there exists a completion J of s^k such that $s_i^n = s_i^k + \sum_{j \in J} v_i^j$ for $i = 1, \ldots, p$ and $s_{p+1}^n = s_{p+1}^k + \sum_{j \in J} w^j$. The set of extensions of s^k is denoted by $Ext(s^k)$ $(k \leq n)$. Finally, $s^k \in S^k$ $(k \leq n)$ is a* restriction *at phase k of state $s^n \in S^n$ if and only if s^n is an extension of s^k.*

3.2 Families of Dominance Relations in Dynamic Programming

The efficiency of DP depends crucially on the possibility of reducing the set of states at each phase. In the context of the approximation version, a family of dominance relations between states for $\underline{\Delta}_\varepsilon$ is used to discard states at any phase. Each dominance relation of this family is specific to a phase. Indeed, we share out the total error ε between the phases by the mean of an error function and associate to each dominance relation of the family a proportion of this error.

Definition 2 (Error function). *The function $e : \{1, \ldots, n\} \to \mathbb{R}$ is an error function if and only if $\sum_{k=1}^{n} e(k) \leq 1$ and $e(k) \geq 0$, $k = 1, \ldots, n$.*

In this way, families of dominance relations between states for $\underline{\Delta}_\varepsilon$ are defined as follows.

Definition 3 (Families of dominance relations between states for $\underline{\Delta}_\varepsilon$). *For any $\varepsilon \geq 0$ and any error function e, a family of relations D^k on S^k, $k = 1, \ldots, n$, is a family of dominance relations for $\underline{\Delta}_\varepsilon$ if for all $s^k, \tilde{s}^k \in S^k$,*

$$s^k D^k \tilde{s}^k \Rightarrow \forall \tilde{s}^n \in Ext(\tilde{s}^k), \exists s^n \in Ext(s^k), s_i^n (1+\varepsilon)^{e(k)} \geq \tilde{s}_i^n, \quad i = 1, \ldots, p \quad (1)$$

When $\varepsilon = 0$, Definition 3 collapses to the classical definition of dominance relations used in the context of the exact version: a relation D^k on S^k, $k = 1, \ldots, n$, is a dominance relation for $\underline{\Delta}$ if for all $s^k, \tilde{s}^k \in S^k$,

$$s^k D^k \tilde{s}^k \Rightarrow \forall \tilde{s}^n \in \mathrm{Ext}(\tilde{s}^k), \exists s^n \in \mathrm{Ext}(s^k), s^n \underline{\Delta} \tilde{s}^n \quad (2)$$

Even if dominance relations can be non-transitive, in order to be efficient in the implementation, we consider only transitive dominance relations. We introduce now the way of using families of transitive dominance relations for $\underline{\Delta}_\varepsilon$ in DP approach (see Algorithm 1). At each phase k, Algorithm 1 generates a subset of states $C^k \subseteq S^k$. This is achieved by first creating from C^{k-1} a temporary subset $T^k \subseteq S^k$. Then, we apply transitive dominance relation D^k to each state of T^k

Algorithm 1. Computing a reduced $(1+\varepsilon)$-approximation

```
1  C^0 ← {(0,...,0)};
2  for k ← 1 to n do
3      T^k ← C^{k-1} ∪ {(s_1^{k-1} + v_1^k, ..., s_p^{k-1} + v_p^k, s_{p+1}^{k-1} + w^k) | s_{p+1}^{k-1} + w^k ≤ W, s^{k-1} ∈ C^{k-1}};
       /* Assume that T^k = {s^{k(1)}, ..., s^{k(r)}}                                   */
4      C^k ← {s^{k(1)}};
5      for i ← 2 to r do
           /* Assume that C^k = {s̃^{k(1)}, ..., s̃^{k(ℓ_i)}}                            */
6          dominated ← false ; dominates ← false ; j ← 1;
7          while j ≤ ℓ_i and not(dominated) and not(dominates) do
8              if s̃^{k(j)} D^k s^{k(i)} then dominated ← true
9              else if s^{k(i)} D^k s̃^{k(j)} then C^k ← C^k \ {s̃^{k(j)}} ; dominates ← true;
10             j ← j + 1;
11         if not(dominated) then
12             while j ≤ ℓ_i do
13                 if s^{k(i)} D^k s̃^{k(j)} then C^k ← C^k \ {s̃^{k(j)}};
14                 j ← j + 1;
15             C^k ← C^k ∪ {s^{k(i)}};
16 return C^n;
```

in order to check if it is not dominated by any state already in C^k (in which case it is added to C^k) and if it dominates states already in C^k (which are then removed from C^k).

The following results characterize the set C^k obtained at the end of each phase k and establish the validity of Algorithm 1.

Proposition 1. *For any transitive relation D^k on S^k, the set C^k obtained at the end of phase k in Algorithm 1 is a covering and independent set of T^k with respect to D^k $(k = 1, \ldots, n)$.*

Theorem 1. *For any family of transitive dominance relations $D^1, \ldots, D^n$ for $\underline{\Delta}_\varepsilon$, Algorithm 1 returns C^n a covering set of S^n with respect to $\underline{\Delta}_\varepsilon$. Moreover, if $\underline{\Delta} \subseteq D^n$, C^n is a reduced $(1+\varepsilon)$-approximation.*

Algorithm 1 can be significantly simplified by generating states of $T^k = \{s^{k(1)}, \ldots, s^{k(r)}\}$ according to any topological order based on the asymmetric part of D^k. Thus, we have either $s^{k(i)} D^k s^{k(j)}$ or not$(s^{k(j)} D^k s^{k(i)})$ for all $i < j$ $(1 \leq i,j \leq r)$ and step 9 and loop 12-14 can be omitted.

Remark that when $\varepsilon = 0$, we have $\underline{\Delta}_\varepsilon = \underline{\Delta}$, and thus C^n is a covering set of X with respect to $\underline{\Delta}$. Moreover, in this case, if $\underline{\Delta} \subseteq D^n$, C^n corresponds to a reduced efficient set.

4 Dominance Relations

We first present the family of dominance relations for $\underline{\Delta}_\varepsilon$ used in our approach that can provide an fptas in certain cases (section 4.1). Then, we present two complementary dominance relations for $\underline{\Delta}$ (section 4.2) and give a brief explanation of the way of applying them together with the family of dominance relations for $\underline{\Delta}_\varepsilon$ (section 4.3).

4.1 Family of Dominance Relations for $\underline{\Delta}_\varepsilon$

To solve the exact version of the 0-1 multi-objective knapsack problem, we showed in a previous work [6] that a powerful dominance relation for $\underline{\Delta}$ is the relation $D^k_{\underline{\Delta}}$ that is a generalization to the multi-objective case of the dominance relation usually attributed to Weingartner and Ness [7]. Relation $D^k_{\underline{\Delta}}$ is defined on S^k for $k = 1, \ldots, n$ by:

$$\text{for all } s^k, \tilde{s}^k \in S^k,\ s^k D^k_{\underline{\Delta}} \tilde{s}^k \Leftrightarrow \begin{cases} s^k \underline{\Delta} \tilde{s}^k & \text{and} \\ s^k_{p+1} \leq \tilde{s}^k_{p+1} \text{ if } k < n \end{cases}$$

To solve the approximation version of the 0-1 multi-objective knapsack problem, we generalize relations $D^k_{\underline{\Delta}}$ $(k = 1, \ldots, n)$ to obtain the family of dominance relations $D^k_{\underline{\Delta}_\varepsilon}$ $(k = 1, \ldots, n)$ for $\underline{\Delta}_\varepsilon$ that is based on a partition of the criterion space into hyper-rectangles. For a constant $\varepsilon > 0$ and an error function e, at each phase k we partition each positive criterion range $[1, U_i]$, where U_i is an upper bound on the value of the feasible solutions on the ith criterion $(i = 1, \ldots, p)$, into disjoints intervals of length $(1+\varepsilon)^{e(k)}$ $(k = 1, \ldots, n)$. When $e(k) = 0$, we obtain the following degenerate intervals: $[1,1], [2,2], \ldots, [U_i, U_i]$ $(i = 1, \ldots, p)$. When $e(k) \neq 0$, we obtain the following intervals: $[1; (1+\varepsilon)^{e(k)}[, [(1+\varepsilon)^{e(k)}; (1+\varepsilon)^{2e(k)}[, \ldots, [(1+\varepsilon)^{(\ell^k_i - 1)e(k)}; (1+\varepsilon)^{\ell^k_i e(k)}[$ where $\ell^k_i = \left\lfloor \frac{\log U_i}{e(k)\log(1+\varepsilon)} \right\rfloor + 1$ $(i = 1, \ldots, p)$. In both cases, we add the interval $[0, 0]$. The number of the interval in which belongs the value of a state s^k on the ith criterion $(i = 1, \ldots, p)$ in this partition is:

$$\mathrm{B}_i(s^k, e(k)) = \begin{cases} s^k_i & \text{if } e(k) = 0 \text{ or } s^k_i = 0 \\ \left\lfloor \frac{\log s^k_i}{e(k)\log(1+\varepsilon)} \right\rfloor + 1 & \text{otherwise} \end{cases}$$

From these partitions, we can define for any $\varepsilon > 0$ and any error function e, relations $D^k_{\underline{\Delta}_\varepsilon}$ on S^k for $k = 1, \ldots, n$ by:

$$\text{for all } s^k, \tilde{s}^k \in S^k,\ s^k D^k_{\underline{\Delta}_\varepsilon} \tilde{s}^k \Leftrightarrow \begin{cases} \mathrm{B}_i(s^k, e(k)) \geq \mathrm{B}_i(\tilde{s}^k, e(k)) \quad i = 1, \ldots, p, \text{ and} \\ s^k_{p+1} \leq \tilde{s}^k_{p+1} \text{ if } k < n \end{cases}$$

The following proposition shows that $D^k_{\underline{\Delta}_\varepsilon}$ is indeed a family of dominance relations for $\underline{\Delta}_\varepsilon$ and gives additional properties of $D^k_{\underline{\Delta}_\varepsilon}$.

Proposition 2. *For any $\varepsilon > 0$ and any error function e, we have:*

(a) $D^k_{\underline{\Delta}_\varepsilon}$, $k = 1, \ldots, n$, is a family of dominance relations for $\underline{\Delta}_\varepsilon$,
(b) for any $k \in \{1, \ldots, n\}$, $D^k_{\underline{\Delta}_\varepsilon}$ is transitive,
(c) for any $k \in \{1, \ldots, n\}$, $D^k_{\underline{\Delta}_\varepsilon} \supseteq D^k_{\underline{\Delta}}$ and $D^k_{\underline{\Delta}_\varepsilon} = D^k_{\underline{\Delta}}$ if $e(k) = 0$.

As a consequence of (c) we have $\underline{\Delta} \subseteq D^n_{\underline{\Delta}_\varepsilon}$ and thus Algorithm 1 using the family of relations $D^k_{\underline{\Delta}_\varepsilon}$, $k = 1, \ldots, n$, computes a reduced $(1+\varepsilon)$-approximation (see Theorem 1). Relation $D^k_{\underline{\Delta}_\varepsilon}$ is a powerful relation since a state can possibly

dominate all other states of larger weight. This relation requires at most $p+1$ tests to be established between two states.

Observe that, even if the authors of [5] do not explicitly mention the use of a family of dominance relations for $\underline{\Delta}_\varepsilon$, their approach could be restated within Algorithm 1 by using the following family of relations D_E^k defined on S^k by:

$$\text{for all } s^k, \tilde{s}^k \in S^k,\ s^k D_E^k \tilde{s}^k \Leftrightarrow \begin{cases} \mathrm{B}_i(s^k, 1/n) = \mathrm{B}_i(\tilde{s}^k, 1/n), & i = 1, \ldots, p, \text{ and} \\ s_{p+1}^k \leq \tilde{s}_{p+1}^k \end{cases}$$

Remark that $D_E^k \subseteq D_{\underline{\Delta}_\varepsilon}^k$ for $e(k) = 1/n$ $(k = 1, \ldots, n)$. This relation, which is quite sufficient to establish the existence of an fptas, has two main disadvantages for an efficient implementation. First, it is very poor since it compares only states lying in the same hyper-rectangle. Therefore, even if two states s^k, $\tilde{s}^k$ are such that $s^k D_{\underline{\Delta}}^k \tilde{s}^k$, we keep both of them in C^k provided that they are not in the same hyper-rectangle. Secondly, by applying a constant error of $(1+\varepsilon)^{1/n}$ at each phase, the total error of $1+\varepsilon$ is shared out equitably among all the phases. During the first phases, since the values of the states are small, the hyper-rectangles in which the states belong usually have a length smaller than 1 on all dimensions. In this case, the advantage of the partition is canceled out since only states with same values could be in relation D_E^k. Thus, the error allocated to these phases is wasted.

For a given $\varepsilon > 0$, the running time of Algorithm 1 using relation $D_{\underline{\Delta}_\varepsilon}^k$ depends crucially on the error function e. In order to guarantee that Algorithm 1 is polynomial both in the size of the instance and in $1/\varepsilon$, we have to add some conditions on the error function aiming at limiting the number of phases with an error equals to 0.

Definition 4 (Polynomial error function). *The error function e is a polynomial error function if for $k = 1, \ldots, n$, $e(k) = 1/g(k)$, if k is a multiple of t, 0 otherwise, where t is a strictly positive integer in $O(\log n)$ and where, for any $k = 1, \ldots, n$, $0 < g(k) \leq cn^d$ for some positive fixed constants c, d.*

The following theorem establishes the complexity of Algorithm 1 using the family of dominance relations $D_{\underline{\Delta}_\varepsilon}^k$.

Theorem 2. *For any $\varepsilon > 0$ and any polynomial error function e, Algorithm 1, using the family of dominance relations $D_{\underline{\Delta}_\varepsilon}^k$, is polynomial both in the size of the instance and in $1/\varepsilon$.*

Hence, by Theorems 1 and 2 we have that, for any $\varepsilon > 0$ and any polynomial error function e, Algorithm 1 using the family of dominance relations $D_{\underline{\Delta}_\varepsilon}^k$ is an fptas that produces a reduced $(1+\varepsilon)$-approximation.

4.2 Complementary Dominance Relations with Respect to $\underline{\Delta}$

Since each dominance relation focuses on specific considerations, it is then desirable to make use of complementary dominance relations. Moreover, when deciding to use a dominance relation, a tradeoff must be made between its potential ability of discarding many states and the time it requires to be checked. We

present now two other complementary dominance relations for $\underline{\Delta}$. The first one, D_r^k, is very easy to establish and the second one, D_b^k, although more difficult to establish, is considered owing to its complementarity with D_r^k and $D_{\underline{\Delta}_\varepsilon}^k$.

Relation D_r^k is based on the following observation. When the residual capacity associated to a state s^k of phase k is greater than or equal to the sum of the weights of the remaining items (items $k+1,\ldots,n$), the only completion of s^k that can possibly lead to an efficient solution is the full completion $J=\{k+1,\ldots,n\}$. It is then unnecessary to generate extensions of s^k that do not contain all the remaining items. We define thus the dominance relation D_r^k on S^k for $k=1,\ldots,n$ by:

$$\text{for all } s^k,\tilde{s}^k \in S^k,\ s^k D_r^k \tilde{s}^k \Leftrightarrow \begin{cases} \tilde{s}^k \in S^{k-1}, \\ s^k = (\tilde{s}_1^k + v_1^k, \ldots, \tilde{s}_p^k + v_p^k, \tilde{s}_{p+1}^k + w^k), \quad \text{and} \\ \tilde{s}_{p+1}^k \leq W - \sum_{j=k}^n w^j \end{cases}$$

This dominance relation is quite poor, since at each phase k it can only appear between a state that does not contain item k and its extension that contains item k. Nevertheless, it is very easy to check since, once the residual capacity $W - \sum_{j=k}^n w^j$ is computed, relation D_r^k requires only one test to be established between two states.

Dominance relation D_b^k is based on the comparison between extensions of a state and an upper bound of all the extensions of another state. In our context, a criterion vector $u=(u_1,\ldots,u_p)$ is an upper bound for a state $s^k \in S^k$ if and only if for all $s^n \in \text{Ext}(s^k)$ we have $u_i \geq s_i^n$, $i=1,\ldots,p$.

We can derive a general type of dominance relations as follows: considering two states $s^k, \tilde{s}^k \in S^k$, if there exists a completion J of s^k and an upper bound $\tilde{u}$ for $\tilde{s}^k$ such that $s_i^k + \sum_{j\in J} v_i^j \geq \tilde{u}_i$, $i=1,\ldots,p$, then s^k dominates $\tilde{s}^k$.

This type of dominance relations can be implemented only for specific completions and upper bounds. In our experiments, we just consider two specific completions J' and J'' defined as follows. Let $\mathcal{O}^i$ be an order induced by considering items according to decreasing order of ratios v_i^k/w^k $(i=1,\ldots,p)$. Let r_i^ℓ be the rank or position of item ℓ in order $\mathcal{O}^i$. Let $\mathcal{O}^{\max}$ be an order according to increasing values of the maximum rank of items in the p orders $\mathcal{O}^i$ $(i=1,\ldots,p)$ and $\mathcal{O}^{sum}$ be an order according to increasing values of the sum of the ranks of items in the p orders $\mathcal{O}^i$ $(i=1,\ldots,p)$. After relabeling items $k+1,\ldots,n$ according to $\mathcal{O}^{\max}$, completion J' is obtained by inserting sequentially the remaining items into the solution provided that the capacity constraint is respected. J'' is defined similarly by relabeling items according to $\mathcal{O}^{sum}$. To compute u, we use the upper bound presented in [8, Th 2.2] computed independently for each criterion value.

Finally, we define D_b^k a particular dominance relation of this general type for $k=1,\ldots,n$ by:

$$\text{for all } s^k,\tilde{s}^k \in S^k,\ s^k D_b^k \tilde{s}^k \Leftrightarrow \begin{cases} s_i^k + \sum_{j\in J'} v_i^j \geq \tilde{u}_i, \quad i=1,\ldots,p \\ \text{or} \\ s_i^k + \sum_{j\in J''} v_i^j \geq \tilde{u}_i, \quad i=1,\ldots,p \end{cases}$$

where $\tilde{u}=(\tilde{u}_1,\ldots,\tilde{u}_p)$ is the upper bound, according to [8, Th 2.2] for $\tilde{s}^k$.

D_b^k is harder to check than relations D_r^k, $D_{\underline{\Delta}}^k$ and $D_{\underline{\Delta}_\varepsilon}^k$ since it requires much more tests and state-dependent information.

4.3 Use of Multiple Dominance Relations

In order to be efficient, we will use the dominance relations $D_r^k, D_{\underline{\Delta}_\varepsilon}^k$, and D_b^k at each phase. As underlined in the previous subsection, dominance relations require more or less computational effort to be checked. Moreover, even if they are partly complementary, it often happens that several relations are valid for a same pair of states. It is thus natural to apply first dominance relations which can be checked easily (such as D_r^k and $D_{\underline{\Delta}_\varepsilon}^k$) and then test on a reduced set of states dominance relations requiring a larger computation time (such as D_b^k).

5 Computational Experiments and Results

5.1 Experimental Design

All experiments presented here were performed on a bi-Xeon 3.4GHz with 3072Mb RAM. All algorithms are written in C++. In the bi-objective case ($p = 2$), the following types of instances were considered:

A) Random instances: $v_1^k \in_R [1, 1000]$, $v_2^k \in_R [1, 1000]$ and $w^k \in_R [1, 1000]$

B) Unconflicting instances, where v_1^k is positively correlated with v_2^k: $v_1^k \in_R [111, 1000]$ and $v_2^k \in_R [v_1^k - 100, v_1^k + 100]$, and $w^k \in_R [1, 1000]$

C) Conflicting instances, where v_1^k and v_2^k are negatively correlated: $v_1^k \in_R [1, 1000]$, $v_2^k \in_R [\max\{900-v_1^k; 1\}, \min\{1100-v_1^k; 1000\}]$, and $w^k \in_R [1, 1000]$

D) Conflicting instances with correlated weight, where v_1^k and v_2^k are negatively correlated, and w^k is positively correlated with v_1^k and v_2^k: $v_1^k \in_R [1, 1000]$, $v_2^k \in_R [\max\{900 - v_1^k; 1\}, \min\{1100 - v_1^k; 1000\}]$, and $w^k \in_R [v_1^k + v_2^k - 200; v_1^k + v_2^k + 200]$.

where $\in_R [a, b]$ denotes uniformly random generated in $[a, b]$. For all these instances, we set $W = \lfloor 1/2 \sum_{k=1}^{n} w^k \rfloor$.

Most of the time in the literature, experiments are made on instances of type A. Sometimes, other instances such as those of type B, which were introduced in [9], are studied. However, instances of type B should be viewed as quasi single-criterion instances since they involve two non conflicting criteria. Nevertheless, in a bi-objective context, considering conflicting criteria is a more appropriate way of modeling real-world situations. For this reason, we introduced instances of types C and D for which criterion values of items are conflicting. In instances of type D, w^k is positively correlated with v_1^k and v_2^k. These instances were introduced in order to verify if positively correlated weight/values instances are harder than uncorrelated weight/values instances as in the single-criterion context [8,10].

For each type of instances and each value of n presented in this study, 10 different instances were generated. In the following, we denote by pTn a p criteria instance of type T with n items. For example, 2A100 denotes a bi-objective instance of type A with 100 items.

In the experiment, we give also, in some tables, the results obtained in [6] by using relations D_r^k, $D_{\underline{\Delta}}^k$ and D_b^k aiming at solving the exact version of the 0-1 multi-objective knapsack problem. These results are denoted by *exact method*.

In the experiment, the approximation method, respectively the exact method, computes only ND_ε, respectively ND. Standard bookkeeping techniques, not considered here, may be used to produce associated solutions.

5.2 Results

We first try to determine the best error function to use in relation $D_{\underline{\Delta}_\varepsilon}^k$. In Table 1 we compare the CPU time in seconds and the size of ND_ε obtained for these three polynomial error functions:

- $e_1(k) = 1/(\lfloor n/t \rfloor)$ if k is a multiple of t, 0 otherwise
- $e_2(k) = \frac{2k/t}{\lfloor n/t \rfloor(\lfloor n/t \rfloor+1)}$ if k is a multiple of t, 0 otherwise
- $e_3(k) = \frac{6(k/t)^2}{\lfloor n/t \rfloor(\lfloor n/t \rfloor+1)(2\lfloor n/t \rfloor+1)}$ if k is a multiple of t, 0 otherwise

where t, which is a strictly positive integer in $O(\log n)$, expresses the frequency of the application of the error. Table 1 shows clearly that the error function has a significant impact on the CPU time and that error function e_2 is significantly better for all types of instances. Thus, in the following, we will use only error function e_2. Observe also that the size of ND_ε is always extremely smaller than the number of non-dominated criterion vectors.

Second, we show the impact of the frequency t in the error function e_2. Table 2 shows that our approach is always faster by setting the frequency $t = \lfloor \log n \rfloor$. Observe that the cardinality of ND_ε is inversely proportional to the frequency t. For example the increase of a factor 3 of the frequency (from $t = \lfloor \log n \rfloor$ to $\lfloor 3 \log n \rfloor$) leads to a decrease of about a factor 3 of the size of ND_ε.

Lastly, we present, in Table 3, the performance of our approach on large size instances. The largest instances solved here are those of type B with 20000 items and the instances with the largest reduced $(1+\varepsilon)$-approximation are those of type D with 900 items. Observe that the average maximum cardinality of C^k, which is a good indicator of the memory storage needed to solve the instances, can be very huge. This explains why we can only solve instances of type D up to 900 items.

Table 1. Impact of different error functions in our approach

type	avg. time in s.			avg. $\lvert ND_\varepsilon \rvert$			exact method [6]	
	e_1	e_2	e_3	e_1	e_2	e_3	avg t. in s.	avg. $\lvert ND \rvert$
2A-400	51.775	34.347	50.022	332.1	199.8	134.3	307.093	4631.8
2B-1000	0.238	0.180	0.299	1.0	1.0	1.0	8.812	157.0
2C-300	74.308	50.265	68.974	615.3	326.4	227.5	373.097	1130.7
2D-150	65.144	47.398	67.758	703.0	384.3	263.1	265.058	3418.5

$\varepsilon = 0.1$ and frequency $t = 1$

Table 2. Impact of the frequency in the error function e_2

type	avg. time in s.				avg. $\lvert ND_\varepsilon \rvert$				exact method [6]	
	$t=1$	$\lfloor \log n \rfloor$	$\lfloor 2 \log n \rfloor$	$\lfloor 3 \log n \rfloor$	$t=1$	$\lfloor \log n \rfloor$	$\lfloor 2 \log n \rfloor$	$\lfloor 3 \log n \rfloor$	avg. t. in s.	avg. $\lvert ND \rvert$
2A-400	34.347	4.536	5.441	7.664	199.8	31.3	16.1	11.9	307.093	4631.8
2B-1000	0.180	0.120	0.406	1.009	1.0	1.3	1.1	1.0	8.812	157.0
2C-300	50.265	7.511	8.084	11.618	326.4	53.6	27.3	18.8	373.097	1130.7
2D-150	47.398	10.874	11.935	16.156	384.3	70.1	35.8	26.4	265.058	3418.5

$\varepsilon = 0.1$ and error function e_2

Table 3. Results of our approach on large size instances

type	n	time in s.			$\lvert ND_\varepsilon \rvert$			avg.
		min	avg.	max	min	avg.	max	$\max_k \{\lvert C^k \rvert\}$
A	100	0.024	0.042	0.072	6	10.1	15	2456.7
	1000	88.121	94.050	103.402	65	68.5	76	321327.6
	2000	896.640	1030.813	1398.060	111	123.9	132	1489132.4
	2500	1635.230	1917.072	2081.410	127	138.6	147	2585169.9
B	1000	0.084	0.118	0.144	1	1.3	2	5596.4
	10000	245.572	269.731	318.808	2	3.3	4	1160906.4
	20000	2424.700	2816.606	3166.580	4	5.3	7	5424849.6
C	100	0.140	0.210	0.316	21	25.3	32	9964.2
	1000	378.135	419.595	471.269	139	150.2	162	923939.4
	2000	3679.770	4296.847	4749.160	255	272.0	285	4256900.6
D	100	1.948	2.356	2.828	50	52.9	57	93507.9
	500	605.837	640.026	681.286	185	196.4	203	3034228.2
	900	4154.610	4689.373	5177.200	297	313.0	329	10276196.8

$\varepsilon = 0.1$, error function e_2 and frequency $t = \lfloor \log n \rfloor$

Table 4. Comparison between the exact method presented in [6] and the approximation method

type	n	exact method [6]		approximation method				
		avg. t. in s.	avg. $\lvert ND \rvert$	avg. t. in s.		avg. $\lvert ND_\varepsilon \rvert$		avg. error
A	100	0.328	159.3	0.042	(÷ 8)	10.1	(÷ 16)	0.0159
	700	5447.921	4814.8	32.275	(÷ 169)	49.5	(÷ 97)	0.0060
B	1000	8.812	157.0	0.118	(÷ 75)	1.3	(÷ 121)	0.0041
	4000	6773.264	1542.3	11.220	(÷ 604)	1.8	(÷ 857)	0.0023
C	100	2.869	558.2	0.210	(÷ 14)	25.3	(÷ 22)	0.0178
	500	4547.978	7112.1	44.368	(÷ 103)	88.3	(÷ 81)	0.0064
D	100	40.866	1765.4	2.356	(÷ 17)	52.9	(÷ 33)	0.0183
	250	3383.545	8154.7	62.970	(÷ 54)	110.9	(÷ 74)	0.0098

Approximation: $\varepsilon = 0.1$, e_2 and frequency $t = \lfloor \log n \rfloor$
The decrease factors of the avg. CPU time and of the size of the returned set, corresponding respectively to avg. t. in s. of exact method / avg. t. in s. of approximation method and $\lvert ND \rvert / \lvert ND_\varepsilon \rvert$, are given into brackets

5.3 Comparison with an Exact Method

The results of a comparative study between the exact method presented in [6] and our approximation method using relations D_r^k, $D_{\underline{\Delta}_\epsilon}^k$, and D_b^k are presented in Table 4. We have selected the method presented in [6] since, as shown in this paper, it is the most effective method currently known.

The two methods have been compared on the same instances and the same computer. Table 4 presents results for instances of type A, B, C, and D for increasing size of n for instances that can be solved by the exact method. We give for each series the "average error" that refers to the measured error *a posteriori*

which is the smallest value of ε such that the returned set is indeed a reduced $(1+\varepsilon)$-approximation.

Considering the CPU time, the approximation method, of course, is always faster than the exact method (up to more than 600 times faster for instances $2B4000$). Observe that, although the cardinality of ND_ε is very small with regard to the cardinality of ND, the quality of the reduced $(1+\varepsilon)$-approximation is very good since for an error *a priori* $\varepsilon = 0.1$, the error *a posteriori* varies from 0.0023 to 0.0183.

6 Conclusion

The purpose of this work was to design a practically efficient fptas, based on a dynamic programming algorithm, for solving the approximation version of the 0-1 multi-objective knapsack problem. We showed indeed that by using several complementary dominance relations, and sharing the error appropriately among the phases, we obtain an fptas which is experimentally extremely efficient.

While we focused in this paper on the 0-1 multi-objective knapsack problem, we could envisage in future research to apply dominance relations based on similar ideas to the approximation version of other multi-objective problems which admit dynamic programming formulations, such as the multi-objective shortest path problem or multi-objective scheduling problems.

References

1. Ehrgott, M.: Multicriteria optimization. LNEMS 491. Springer, Berlin (2005)
2. Papadimitriou, C.H., Yannakakis, M.: On the approximability of trade-offs and optimal access of web sources. In: IEEE Symposium on Foundations of Computer Science, pp. 86–92 (2000)
3. Safer, H.M., Orlin, J.B.: Fast approximation schemes for multi-criteria combinatorial optimization. Working Paper 3756-95, Sloan School (1995)
4. Safer, H.M., Orlin, J.B.: Fast approximation schemes for multi-criteria flow, knapsack, and scheduling problems. Working Paper 3757-95, Sloan School (1995)
5. Erlebach, T., Kellerer, H., Pferschy, U.: Approximating multiobjective knapsack problems. Management Science 48(12), 1603–1612 (2002)
6. Bazgan, C., Hugot, H., Vanderpooten, D.: An efficient implementation for the 0-1 multi-objective knapsack problem. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 406–419. Springer, Heidelberg (2007)
7. Weignartner, H., Ness, D.: Methods for the solution of the multi-dimensional 0/1 knapsack problem. Operations Research 15(1), 83–103 (1967)
8. Martello, S., Toth, P.: Knapsack Problems. Wiley, New York (1990)
9. Captivo, M.E., Climaco, J., Figueira, J., Martins, E., Santos, J.L.: Solving bicriteria 0-1 knapsack problems using a labeling algorithm. Computers and Operations Research 30(12), 1865–1886 (2003)
10. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Berlin (2004)

Solutions to Real-World Instances of PSPACE-Complete Stacking

Felix G. König, Macro Lübbecke, Rolf Möhring, Guido Schäfer*, and Ines Spenke*

Technische Universität Berlin, Institut für Mathematik, MA 6-1
Straße des 17. Juni 136, 10623 Berlin, Germany
{fkoenig,m.luebbecke,moehring,schaefer,spenke}@math.tu-berlin.de

Abstract. We investigate a complex stacking problem that stems from storage planning of steel slabs in integrated steel production. Besides the practical importance of such stacking tasks, they are appealing from a theoretical point of view. We show that already a simple version of our stacking problem is PSPACE-complete. Thus, fast algorithms for computing provably good solutions as they are required for practical purposes raise various algorithmic challenges. We describe an algorithm that computes solutions within 5/4 of optimality for all our real-world test instances. The basic idea is a search in an exponential state space that is guided by a state-valuation function. The algorithm is extremely fast and solves practical instances within a few seconds. We assess the quality of our solutions by computing instance-dependent lower bounds from a combinatorial relaxation formulated as mixed integer program. To the best of our knowledge, this is the first approach that provides quality guarantees for such problems.

1 Introduction

In this paper, we investigate a dynamic stacking problem that has applications in many production processes with intermediate storage for bulky items. Our work is motivated by a cooperation with PSI-BT [14], a software company that develops planning software for logistics and production processes in steel plants. A crucial task in such a plant is the transportation and intermediate storage of steel slabs over time, which is the prototype of the general stacking problem considered here. It similarly occurs in other settings such as the shunting of rail cars or container stowage.

We keep our problem formulation general enough to provide a concise, theoretically interesting problem class, but still specific enough to be applicable to different industrial scenarios.

In practice, many different constraints like precedence relations, time windows, stack heights, etc. have to be respected, and it is not surprising that the literature abounds with heuristics.

* Supported by the DFG Research Center MATHEON "Mathematics for key technologies."

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 729–740, 2007.

However, no general and versatile problem formulation like ours has been presented before, let alone, solutions to them of proven quality.

From a computer science point of view, our problem is related to motion planning and sorting with networks of stacks. The *Towers of Hanoi* puzzle is just a very simple instance of our model. More interestingly, there are similarities to the much more complex *block sliding* puzzles. Many of these puzzles are known to be PSPACE-complete [8]. We show that this is true also for our stacking problem, which rules out the existence of efficient algorithms in general.

Nevertheless, using ideas from dynamic programming and mixed integer programming, well-established techniques from discrete optimization, we compute solutions to all our real-world instances within 5/4 of an optimum.

Motivation. In integrated steel production, steel is casted in a 24/7 operation in slabs, bars of up to 12 meters length and 30 tons weight. These are rolled to sheet metal in a following batch process. Each slab is assigned to a customer order before it is even casted, and is, thus, individual. The rolling at later points in time is in a given order which (also because of technical reasons) typically differs considerably from the casting sequence. In between, slabs are brought by cranes to a storage area where they are piled up on stacks (for up to several days). If slabs are needed much later (several weeks), they may leave the so-called *hot-buffer* and can be temporarily stored in a much larger *cold-buffer*. In both cases, the number of stacks and their height is limited.

In the hot-buffer, ideally, slabs should be stored in such a way that they are readily available according to the planned batch sequence of the rolling process (each batch may request several dozens of slabs in a given order). This conflicts with the limited space, the casting sequence, and the complicating fact, that slabs exiting a strand caster have to be moved away within tight time windows. Time is less crucial in the cold-buffer, but the system inherent non-conformity of input and output sequences becomes much more visible. Obviously, the buffers constitute a major bottleneck, and a high productivity is desired.

Our Contribution. The STACKING PROBLEM we describe captures practical stacking processes accurately, yet it defines a general and versatile problem type. We show that it is PSPACE-complete, even in a quite basic version. It is therefore unlikely that there exists an efficient algorithm to compute solutions of a proven quality for this problem.

We define a state graph for practical stacking problems and develop a state-valuation function which guides a partial exploration of this typically exponential graph on all feasible stackings. The result is a kind of dynamic programming algorithm which computes good solutions for real-world stacking instances fast.

The power of this approach is that it provides sufficient flexibility to model all constraints that are relevant from a practical point of view and at the same time also allows us to use different techniques and heuristics to speed-up the state space exploration. We also report on the effect of a rollout strategy [2], which gives slightly improved solutions at the expense of a considerable increase in running time.

This is the first time that a proof of the solution's quality for a stacking problem of this versatility is given. We do so by computing an instance-dependent lower bound using a mixed integer program. On our industrial data we obtain solutions that are less than 25% off optimum, usually considerably better.

Related Work. Practical stacking problems have been addressed in the literature a lot, like in steel production [7,13], container terminals [5,12], in the very general area of generic planning problems [6], and several more. All of these papers significantly simplify the problems, in particular the precedence constraints implied by stacking. Methodologies applied range from simulated annealing [7], simulations [5], genetic algorithms [13], state space evaluations [3,10], but, to the best of our knowledge, no principally exact approaches have been suggested.

2 Problem Definition and Hardness Results

We formally describe the STACKING PROBLEM, omitting a few practically relevant details; we comment on these where appropriate. Let $[n] := \{1, \ldots, n\}$.

Incoming items. We have a set $I := [n]$ of incoming items that arrive on m parallel inputs over time. Each item $i \in I$ is associated with a *time window* $[r_i, d_i] \subseteq \mathbb{R}^+$; r_i is the *release time* and d_i the *due time* of item i. An item i must be removed from the input within its time window. We may either move it to one of the k *buffer stacks* or directly to one of the *target stacks* defined below. We assume that at any time at most one item is available at every input.

Buffer stacks. Let $\mathcal{S} := \{S_1, \ldots, S_k\}$ be a set of k *buffer stacks*. Stack $S \in \mathcal{S}$ can hold up to $h(S)$ items. We write $i \rightsquigarrow_S j$ iff i lies, not necessarily directly, on top of j in S. We use the same notation for the target stacks defined below. An allocation of items to stacks $C := (S_1, \ldots, S_k)$ with their respective positions is called a *configuration*. The initial configuration C_0 need not be empty. The set of items that are allocated to the buffer stacks in C_0 is denoted by J. The entire set of items is thus $V := I \cup J$.

Stacking constraints. Items may not be placed arbitrarily on top of each other. We model stacking restrictions by a *conflict graph*. Let $G := (V, A)$ be a directed graph with vertex set V and arc set A. Item $i \in V$ cannot be placed *directly* on top of item j iff $(i, j) \in A$. We call a configuration $C = (S_1, \ldots, S_k)$ *feasible* if every buffer stack $S \in \mathcal{S}$ contains at most $h(S)$ items and for all $i, j \in V$ such that i lies directly on top of j, we have $(i, j) \notin A$. We assume that the initial configuration C_0 is feasible.

Target stacks. We are given a set $\mathcal{T} := \{T_1, T_2, \ldots, T_\ell\}$ of *target stacks*. Each target stack $T_i \in \mathcal{T}$ specifies an order (from first to last) in which the respective items have to be collected. Every item $i \in V$ occurs in at most one target stack; define $t_i \in \mathcal{T}$ as i's target stack and let $t_i := \emptyset$ if no such target stack exists. Items for no more than w target stacks may be collected in parallel, capturing the notion of limited space in the exit area of the buffer.

Once we have started collecting items for a target stack, we may only move away, or *dispose*, this target stack when all of its items have been collected. Additionally, the order in which we dispose target stacks must obey precedence constraints defined by a partial order $\prec_{\mathcal{T}}$ on $\mathcal{T}$: if $T_1 \prec_{\mathcal{T}} T_2$ for target stacks $T_1, T_2 \in \mathcal{T}$ then T_2 can only be disposed after T_1 is disposed.

Moves and objective. We have four possible kinds of moves; (a) an item can be moved from an input to a buffer stack, (b) from an input to its target stack (a *direct move*), (c) from the top of a buffer stack to the top of another buffer stack (this is called *restacking*), and (d) from the top of a buffer stack to a target stack.

A move is feasible if it respects all the conditions like time windows, height bounds, stacking conflicts, accessibility of target stacks, etc. defined above. Transport times are known, but important for feasibility only, since they are small as compared to pickup and drop-off times for items. Thus, our goal is to build all target stacks with a minimum number of feasible moves, starting from the initial configuration C_0. Naturally, this includes determining the exact order in which target stacks are started and disposed. The *feasibility version* asks whether some feasible sequence of moves exists to build all target stacks.

Theorem 1. *The* STACKING PROBLEM *is in PSPACE.*

Proof. We exploit Savitch's Theorem [11] which states the equivalence between the complexity classes PSPACE and NPSPACE. We can represent any configuration of the STACKING PROBLEM in polynomial space. Moreover, all possible moves from a given configuration can be computed in polynomial time. We can therefore perform a nondeterministic search using only polynomial space: In each step we choose one of the possible moves nondeterministically and only keep track of the current state. Savitch's Theorem states that any such nondeterministic PSPACE algorithm can be transformed into a deterministic one. □

We show that the feasibility version of the STACKING PROBLEM is PSPACE-complete using the *nondeterministic constraint logic model of computation (NCL)* introduced by Hearn and Demaine [8]. The NCL model is an alternative view on the complexity class PSPACE. It does not need to adopt a two-player view (like in reductions from QUANTIFIED BOOLEAN FORMULAS), and is thus much better suited to our problem.

More formally, we are given an undirected graph $\hat{G} = (\hat{N}, \hat{E})$ with non-negative integer weights on the edges and integral minimum inflow constraints for the nodes. A *configuration* of $\hat{G}$ corresponds to an orientation of the edges such that for every node the sum of the weights of all incoming edges is at least the minimum inflow constraint of that node. A move from one configuration to another corresponds to the reversal of a single edge such that all inflow constraints remain satisfied. We consider the decision problem whether, starting from a given initial configuration $\hat{C}_0$, there exists a sequence of moves such that a designated edge can be reversed. This problem is referred to as CONFIGURATION-TO-EDGE. It is PSPACE-complete even if $\hat{G}$ is a planar AND/OR graph, i.e., consists of the very simple OR and AND nodes only, depicted in Fig. 1 (a) and (b).

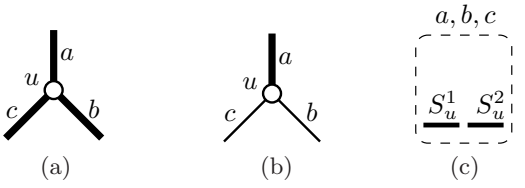


Fig. 1. Illustration of an OR (a) and AND (b) node. Bold edges have weight two, light edges have weight one; the minimum inflow constraint is two. For the OR node, one edge can be directed outward iff at least one of the other two edges is directed inward. For the AND node, edge a may be directed outward iff b and c are directed inward. The gadget for both nodes is depicted in (c).

Theorem 2. *The feasibility version of the* STACKING PROBLEM *is PSPACE-complete, even if there are no incoming items, each buffer stack can hold at most three items, and only one item is requested at a target stack.*

Proof. We present a polynomial-time reduction from CONFIGURATION-TO-EDGE with an AND/OR graph $\hat{G} = (\hat{N}, \hat{E})$ to our STACKING PROBLEM. To this end, we construct AND and OR gadgets that consist of two buffer stacks each. Abusing notation slightly, we have one item $e \in I$ for every edge $e \in \hat{E}$, and every buffer stack $S \in \mathcal{S}$ can hold at most two or three items. We consider an edge e directed outward a node u iff item e is placed on the buffer stacks corresponding to u.

Consider an OR node $u \in \hat{N}$, see Fig. 1(a). Our OR gadget consists of two buffer stacks S_u^1 and S_u^2 on which only the items a, b or c can be placed, see Fig. 1(c). We enforce this restriction by placing dummy items z_u^1 and z_u^2 at the bottom of S_u^1 and S_u^2, respectively, and adding arcs (x, z_u^1) and (x, z_u^2) for all $x \notin \{a, b, c\}$ to our stacking conflict graph. Imposing a height constraint of two, we can place at most two items in $\{a, b, c\}$ on these stacks. Edge $x \in \{a, b, c\}$ is directed outward of u iff x is placed on S_u^1 or S_u^2, that is, at least one edge must be directed inward. Clearly, this gadget implements an OR node.

Next consider an AND node $u \in \hat{N}$, with incident edges a, b and c, see Fig. 1(b). Our AND gadget again consists of two buffer stacks S_u^1 and S_u^2. Again with the help of dummy items, we enforce the following placement restrictions. There are two helper items h_a and h_b which can be placed on top of each other, and at the bottom of S_u^1 or S_u^2. The heavy item a can be placed only at the bottom of S_u^1 or S_u^2. The light items a and b can be placed only at their corresponding helper item h_a and h_b, respectively. A height bound of three on S_u^1 and S_u^2 now guarantees that edge a can be directed outward only, if edges b and c are directed inward. One easily checks that exactly all feasible edge configurations are represented by a feasible item configuration, so this gadget implements an AND node.

For some initial configuration $\hat{C}_0$ and a designated edge $e \in \hat{E}$, CONFIGURATION-TO-EDGE now reduces to the decision problem whether, starting from the buffer stack configuration corresponding to $\hat{C}_0$, there exists a sequence of moves such that the item e can eventually be moved. This decision problem can be simulated by letting the dummy item that is hidden by e in the initial buffer stack configuration be the only item that is requested at the target stack. □

In the PSPACE-completeness proof above we exploit crucially that complex restacking operations may be necessary in order to access a particular item. The key ingredients seem to be that we start with a non-empty initial configuration and that every item has only a very limited number of buffer stacks onto which it can be placed, i.e., the stacking conflict graph is rather dense. The next theorem shows that the problem remains hard even if we remove these assumptions.

Theorem 3. *The problem of deciding whether a given target sequence can be built without any restacking operations is NP-hard if all buffer stacks have a uniform height bound of $h \geq 6$, even if we start with empty buffer stacks and there are no conflicts between items.*

Proof. We reduce MUTUAL EXCLUSION SCHEDULING [9] for permutation graphs to our STACKING PROBLEM. In this problem, we are given a permutation graph $\hat{G}(\Pi) = (\hat{N}, \hat{E})$ of n nodes and a positive integer h. Jansen [9] proved that for every fixed $h \geq 6$, the decision problem whether there exists a partition of $\hat{N}$ into t independent sets of size at most h is NP-hard.

Let $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ be the permutation of $\langle 1, \ldots, n \rangle$ that defines the permutation graph $\hat{G}(\Pi)$. We define an instance of our STACKING PROBLEM as follows: Items appear at the input in the order $\langle \pi_1, \ldots, \pi_n, n+1 \rangle$ and are requested at the target stack in the order $T = \langle n+1, n, \ldots, 1 \rangle$. The buffer stacks are empty initially and have the same height bound h. It is not difficult to verify that the target stack T can be built without any restacking operations, i.e., using $2n+1$ moves, iff $\hat{N}$ can be partitioned into $t = k$ independent sets of size at most h. □

3 Guided Graph Search

We use graph terminology to specify the state space which is searched for a solution to the STACKING PROBLEM: We associate a node with each possible buffer configuration C and define the neighborhood of a node as the set of all configurations that can be constructed from C by one feasible move. The notion of feasibility of a move includes that *after* the move it must still be possible to remove all items from the inputs respecting their time windows. The due time to remove items from their respective input in order to be able to feasibly serve all inputs can be determined by a simple backward-calculation.

Note that the actual nodes of the state graph do not only consist of a buffer configuration C but also comprise a time stamp t and a notion of progress made. This progress includes the items already moved from the inputs, say A, and the set of target stacks for which items are currently being collected Ω. We call $\Sigma := (C, t, A, \Omega)$ a *state*; every state corresponds to exactly one node in the state graph.

Obviously, it is impossible to generate even only a significant part of the state graph—already the number of possible buffer configurations is huge. Thus we use a *valuation function* on the buffer states to tell us which parts of the graph are "interesting". This approach is related to the idea of approximate cost-to-go functions in dynamic programming [1].

The definition of a proper valuation function depends on the nature of the instances to be solved. We give two examples in Sections 3.1 and 3.2. The first one yields an optimal algorithm for the Towers of Hanoi problem. The latter one leads to good solutions for instances of our real-world application and may well be suitable for other applications involving the storage of items on stacks.

The power of this approach is two-fold: On the problem formulation side, all constraints regarding the structure of the stacks, travel times and the necessity to respect time windows when removing items from the inputs can easily be modeled by modifying the rules with which a node's neighborhood is created; on the solution side, different techniques can be used for the exploration of the graph allowing fine-tuning of the algorithm towards speed, robustness, precision, flexibility or other goals.

3.1 The Towers of Hanoi Example

We formulate the Towers of Hanoi problem as a STACKING PROBLEM and solve it optimally by our algorithm using a suitable valuation function. In this case, there are no arriving items and no constraints for the number and height of the stacks; only stacking constraints have to be respected.

There are n discs $0, 1, \ldots, n-1$ with diameters $d(0) > d(1) > \cdots > d(n-1)$ and a larger disc may never be stacked on a smaller one. In the beginning, all discs are stacked feasibly on S_1 and the goal is to stack all of them feasibly on S_3 by only moving one disc at a time.

Then following iterative algorithm leads to the unique optimal solution [15]. First, arrange the three stacks S_1 (source), S_2, and S_3 (target) into a triangle with S_1 on the left, S_2 on top and S_3 on the right. Apply two rules:

($R1$) If n is even, move even discs clockwise and odd discs counter-clockwise only; if n is odd, do it vice versa.
($R2$) In move k, starting at 1, move the disc corresponding to the power of the right-most (i.e., least-significant) 1-bit in the binary representation of k.

These two rules imply the following two facts; we omit their proofs.

($F1$) In the unique optimal solution, a disc i is never moved to a non-empty stack S_ℓ, such that the difference between i and the top-most disc j in S_ℓ is even.
($F2$) In the unique optimal solution, a disc is only moved to an empty stack if every other feasible move violates Fact ($F1$).

We use ($F1$) and ($F2$) to define a valuation function. Let $e(C)$ be the number of empty stacks in a configuration C and $v(S_\ell) = 1$ if the difference between the two topmost discs in stack S_ℓ is even, and 0 otherwise. Let $w > 1$ be an arbitrary number making $e(C)$ a tie-breaker enforcing Fact ($F2$).

$$val^{hanoi}(C) := w \cdot \left(\sum_{\ell=1}^{3} v(S_\ell) \right) - e(C) \tag{1}$$

Let C_1 and C_2 be two neighbored configurations in the unique optimal sequence of moves. Then $val^{hanoi}(C_2)$ is the unique minimum among all $val^{hanoi}(C)$, where C is reached from C_1 by a feasible move. Applying our greedy graph search we thus obtain the unique optimal sequence of moves.

It is easy to see that time and space required by our search are of the same order as the known lower bounds (see [4]) of $\Theta(2^n)$ and $\Theta(n)$, respectively.

3.2 The Application-Driven Greedy Approach

Given a buffer configuration $C = (S_1, \ldots, S_k)$ and the set of target stacks $\mathcal{T}$ to be compiled, there is a natural lower bound on the number of moves needed: We count one move for each item that will be moved from a buffer to a target stack plus one move for each item which has to be moved out of the way; we will call the latter the number of *false positions* in C.

In order to know more precisely which items must be "moved out of the way", we determine in a preprocessing step a linear extension $<_{\mathcal{T}}$ of the partial order $\prec_{\mathcal{T}}$ in which we will begin collecting items for target stacks. This is done by a simple heuristic, which first orders all target stacks according to the latest release time of one of their items, and then ensures $T_i <_{\mathcal{T}} T_j$ for all T_i, T_j with $T_i \prec_{\mathcal{T}} T_j$ by delaying T_j until directly after T_i in $<_{\mathcal{T}}$ if necessary. Now assume without loss of generality that $T_1 <_{\mathcal{T}} T_2 <_{\mathcal{T}} \ldots$

We now define the following partial precedence order on the items: For two items i, j, we write $i \prec j$ iff $t_i <_{\mathcal{T}} t_j$ and $\{t_i, t_j\} \not\subseteq \Omega$, or $t_i = t_j$ and $j \rightsquigarrow_{t_i} i$. The number of false positions for item i on stack S is then

$$\mathit{false}(i, S) := |\{j \in S \ : \ j \rightsquigarrow_S i \ \wedge \ i \prec j\}| \ . \tag{2}$$

From any configuration C without false positions, we can build all required target stacks without a single restacking operation. Thus, we define a valuation function which deems those buffer configurations interesting which have few false positions.

We introduce a valuation functions val^{app} on states $\Sigma = (C, t, A, \Omega)$:

$$val^{app}(\Sigma) := \sum_{\theta=\gamma}^{|\mathcal{T}|} d^{\gamma-\theta} \cdot val^{app}_{\theta}(\Sigma), \tag{3}$$

where where γ is the smallest index of a target stack not yet disposed and $d \leq 1$ is some discount factor determining how fast the weight of false positions decreases for target stacks to be compiled in the future,

$$val^{app}_{\theta}(\Sigma) := \sum_{\ell=1}^{k} \sum_{i \in S_\ell \cap T_\theta} \mathit{false}(i, S_\ell) \qquad \forall \theta = \gamma, \ldots, |\mathcal{T}| \tag{4}$$

accounts for the total number of false positions for items in T_θ.

Note that the sum of all false position in a configuration constitutes a lower bound on the number of necessary moves only with respect to $<_{\mathcal{T}}$. Theoretically,

starting to collect items for target stacks in a different order may lead to a better solution. Yet we have found that the number of false positions computed with respect to $\prec_{\mathcal{T}}$ only is too weak of a bound to be of any help in the state space search.

In view of some uncertainty in related processes and the strict bounds on computation time prescribed by the slab stacking application, the most important features of the used algorithm are speed and the ability to react to changes in data. The latter can naturally be achieved by defining the state of the buffer after a change in data as a new initial state and rerunning the algorithm, making the speed of the algorithm an even more important goal for practical purposes.

Thus, a fast greedy variant of the graph search procedure is used, which in each current node of the state graph computes the valuation function for all neighboring nodes and picks the one with best value. To break ties, we prefer moves which take less time and try to maintain as many unused stacks in the buffer as possible. Somewhat surprisingly, this simple and extremely fast search strategy leads to good solutions to practical instances, as described in Section 5.

An additional parameter, the *lookahead* f, is introduced to speed up the calculation of the valuation function for each buffer state, especially for instances where data is available for many future target stacks: Instead of considering *all* values val_θ^{app} in (3), we only take the first $f+1$ values into account; i.e., we replace $|\mathcal{T}|$ by $\gamma + f$ in (3). The effects of different choices for f for the practical instances are described in Section 5.

Note that this valuation function favors buffer states in which items can be moved to a target stack, but does not yet encourage the algorithm to actually do so. Thus, we introduce an additional rule for the choice of a neighbor, always preferring a state which moves an item to a target stack over one that does not. The order in which the algorithm starts to build the target stacks is computed in a preprocessing step by the following simple rule: Build those target stacks first, for which all items have arrived at the buffer the earliest while respecting precedence constraints on the target stacks where necessary.

In Section 5, we also report on the effects of combining our greedy algorithm with a rollout strategy [2]. Due to space limitations, we only sketch the basic idea here: If we have reached a state Σ in our state graph, we run the greedy algorithm from each neighboring state and then continue with the state that returns the mimimum number of moves. Clearly, this approach entails computational overhead, but may improve the solution quality.

4 Lower Bounds from a Combinatorial Relaxation

In order to assess the quality of our solutions we compute lower bounds on the optimal number of moves per instance. If the initial configuration is non-empty, one may count how many items on top of an item are needed later and thus have to be moved away. This bound is used for the cold-buffer instances below.

A more elaborate technique is needed for the hot-buffer instances. We formulate a mixed integer program (MIP) and derive a bound from its linear

programming (LP) relaxation. However, an MIP which captures the STACKING PROBLEM in full detail is far from being computationally tractable. In fact, in order to represent an exponential solution one would have to work explicitly with an exponential number of variables.

Instead, we devise a non-obvious but surprisingly useful *combinatorial* relaxation of the problem, which is mainly based on the time windows at the inputs: We assume a sufficiently large number $k \geq n$ of stacks. This implies that we need not take care of infeasible buffer configurations because each item can use its own stack. As a consequence, we disregard the concept of stacks and conflicts at all, except for target stacks. The objective is to maximize the number of direct moves from the inputs to the target stacks. For this relaxation, the MIP computes a best possible linear extension $<_{\mathcal{T}}$ of the partial order $\prec_{\mathcal{T}}$ in which to begin collecting items for target stacks.

Even though the MIP is deprived the essential stacking character, it turns out that the LP relaxation gives a good lower bound in practice, see the hot-buffer instances in Section 5.

5 Computational Results

We tested two different sets of industrial instances supplied by PSI-BT: *hot-buffer instances* with few buffer stacks and tight time windows and *cold-buffer instances* with many more buffer stacks, but without incoming items. In both cases, all target stacks need to be built with a minimum number of moves. All experiments were performed on a standard PC running Linux.

Hot-buffer instances. The time horizon for these instances is up to 12 hours. All instances have 3 strand caster inputs, 3 target stacks can be built in parallel. The number of buffer stacks varies between 14 and 16. The crane can transport one item at a time. The numbers of needed moves are stated in Table 1.

Table 1. Solution quality for the hot-buffer instances of the greedy algorithm (and rollout enhancement). LB refers to the MIP lower bound described in Section 4.

	v1	v2a	v2b	v3
LB	325	496	251	732
greedy	362	561	264	794
gap (%)	10.2	11.6	4.9	7.8
rollout	359	538	260	785
gap (%)	9.5	7.8	3.5	6.8

The runtime for the greedy algorithm is less than 0.1 seconds per move. The optimality gap is below 12% and can be improved to less than 10% using a rollout strategy. An optimal integer solution to the MIP was computed using the commercial solver CPLEX 10.1 within some minutes to a few hours. No integer solution was found for instances v3 and we used the LP bound instead.

Unfortunately, no data about the crane transports that are nowadays performed by the manual operators is available. However, the steel plant states that at least about half of all transports are restacking operations, while in our solutions they account to less than 20%. This suggests the buffer performance could be greatly improved by the implementation of our algorithm in practice.

Cold-buffer instances. All instances have 50 buffer stacks, and 5 target stacks can be built simultaneously. There are no incoming items and therefore our MIP lower bound does not apply. The crane can transport several items, up to a maximum weight limit, simultaneously. Again, numbers of needed moves are given in Table 2.

Table 2. Solution quality for the cold-buffer instances of the greedy algorithm (and rollout enhancement). LB is the simple lower bound counting false positions.

	aa1	aa2	aa3	aa4	ab1	ab2	ab3	ab4	ab5	ab6	ab7	ab8	ab9	ab10
LB	30	52	109	109	105	111	108	103	104	103	110	110	113	104
greedy	36	64	128	126	131	135	126	125	119	126	129	130	129	118
gap (%)	20.0	23.1	17.4	15.6	24.8	21.6	16.7	21.4	14.4	22.3	17.3	18.2	14.2	13.5
rollout	34	60	119	120	124	125	117	114	115	113	122	122	120	113
gap (%)	13.3	15.4	9.2	10.1	18.1	12.6	8.3	10.7	10.6	9.7	10.9	10.9	6.2	8.7

For these instances, more than 50% of the overall moves are restacking moves. Our algorithm needs less than 3 seconds per move. The solution quality is below 25%, despite the large share of restacking moves.

In all our experiments, the computational overhead produced by the rollout strategy was enourmous (increase in runtime by a factor of more than 1000), while the gain in quality is marginal—the application of this method in practice is therefore not conceivable.

6 Discussion

A major difficulty in developing better lower bounds is to capture the problem-inherent restacking operations.

The Towers of Hanoi example shows that exponentially many restacking operations may be needed; also our PSPACE-completeness proof relies crucially on these operations. On the other hand, in our practical instances we observed only a rather small number of restacking moves. It would be very interesting to characterize the hardness of an instance by means of its "restacking complexity". Yet, in an ongoing work, we have experienced that such a characterization is not obvious at all, even for two buffer stacks only.

One may argue that a solution quality of 25% off optimum is rather weak. Three comments are in order here. First, we conjecture that the quality of the computed solutions is much better than what we are able to prove. Second, in the area of approximation algorithms, an approximation factor of 5/4 is considered

to be very good for a problem that does certainly not admit a PTAS. Third, in recent years we have witnessed tremendous advances in the area of computational combinatorial optimization, in particular mixed integer programming. We believe that striving for close-to-optimal solutions is a necessary and fruitful step in advancing the field even further.

We hope that our results encourage further investigations concerning the applicability of discrete optimization to PSPACE-complete problems. The practical relevance of these approaches in industry is evident. In fact, a prototype implementation of our algorithm is used to evaluate the buffer performance of two steel plants already. Animated visualizations of our stacking plans were shown to several practitioners, who were quite impressed by the low share of restacking operations and the anticipation of slabs needed on target stacks in the future.

References

1. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Athena Scientific (1996)
2. Bertsekas, D.P., Tsitsiklis, J.N., Wu, C.: Rollout algorithms for combinatorial optimization. Journal of Heuristics 3(3), 245–262 (1997)
3. Bonet, B., Loerincs, G., Geffner, H.: A robust and fast action selection mechanism for planning. In: Proceedings of the 14th National Conference on AI, pp. 714–719 (1997)
4. Cull, P., Ecklund, E.F.: Towers of hanoi and analysis of algorithms. The American Mathematical Monthly 90, 407–420 (1985)
5. Dekker, R., Voogd, P., van Asperen, E.: Advanced methods for container stacking. OR Spectrum 28, 563–586 (2006)
6. Fox, M., Long, D.: Progress in AI planning research and application. CEPIS 3, 10–25 (2002)
7. Hansen, J., Clausen, J.: Crane scheduling for a plate storage. Technical Report 1, Informatics and Mathematical Modelling, Technical University of Denmark (2002)
8. Hearn, R.A., Demaine, E.D.: PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. Theoretical Computer Science 343(1–2), 72–96 (2005)
9. Jansen, K.: The mutual exclusion scheduling problem for permutation and comparability graphs. Information and Computation 180, 71–81 (2003)
10. McDermott, D.: A heuristic estimator for means ends analysis in planning. In: Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems, pp. 142–149 (1996)
11. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. Journal of Computer and System Sciences 4(2), 177–192 (1970)
12. Steenken, D., Voß, S., Stahlbock, R.: Container terminal operation and operations research - a classification and literature review. OR Spectrum 26, 3–49 (2004)
13. Tang, L., Liu, J., Rong, A., Yang, Z.: Modelling and a genetic algorithm solution to the slab stack shuffling problem in implementing steel rolling schedulings. International Journal of Production Research 40(7), 1583–1595 (2002)
14. http://www.psi-bt.com
15. Walsh, T.: The towers of hanoi revisited, moving the rings by counting the moves. Information Processing Letters 15(2), 64–67 (1982)

Non-clairvoyant Batch Sets Scheduling: Fairness Is Fair Enough

Julien Robert[1] and Nicolas Schabanel[2]

[1] École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France
http://perso.ens-lyon.fr/julien.robert
[2] CNRS Centro de Modelamiento Matemático, Blanco Encalada 2120 Piso 7, Santiago de Chile
http://www.cmm.uchile.fr/∼schabanel

Abstract. In real systems, such as operating systems, the scheduler is often unaware of the remaining work in each job or of the ability of the job to take advantage of more resources. In this paper, we adopt the setting for non-clairvoyance of [3,2]. Based on the particular case of malleable jobs, it is generally assumed in the literature that "**Equi** *never starves a job since it allocates to every job the same amount of processing power*". We provide an analysis of the competitiveness of **Equi** for the makespan objective which shows that under this more general setting this statement is at the same time true and false: false, because, some jobs may be stretched by a factor as large as, but no more than, $\frac{\ln n}{\ln \ln n}$ with respect to the optimal, where n is the size of the largest set; true, because no algorithm can achieve a better competitive ratio up to a constant factor.

In this paper, we extend the results in [2,11] to the batch scheduling of sets of jobs that go through *arbitrary* phases: user request all together at time 0, for the execution of a set of jobs and is served when the last job completes. We prove that the algorithm **Equi**∘**Equi** is $(2+\sqrt{3}+o(1))\frac{\ln n}{\ln \ln n}$-competitive, where n is the maximum size of a set, which is optimal up to a constant factor. We provide experimental evidences that this algorithm may have the same asymptotic competitive ratio $\Theta(\frac{\ln n}{\ln \ln n})$ (independent of the number of requests) for the flowtime objective when requests have release dates, if it is given sufficiently large extra processing power with respect to the optimum.

Keywords: Online scheduling, Non-clairvoyant algorithm, Batch scheduling, Fairness, Equi-partition, Makespan and Overall Set Completion Time minimization.

1 Introduction

Scheduling questions arise naturally in many different areas among which operating system design, compiling, memory management, communication network, parallel machines, clusters management,... In real systems, the characteristics of the jobs to schedule (such as release time, processing time,...) are often unknown

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 741–753, 2007.

and/or unpredictable beforehand. In particular, the scheduler, such as in an operating system, is typically unaware of the remaining work in each job or of the ability of the job to take advantage of more resources. Such systems are referred as *non-clairvoyant*.

Several settings have been proposed to model non-clairvoyance, in which jobs are fully parallelizable but their amount of work is unknown before their completion [9,6,7,5,8,1]. In this paper, we consider the very general setting for non-clairvoyance proposed by Edmonds [3,2]. Jobs go through a sequence of different phases, each consisting of a certain quantity of work with a speed-up function that quantifies how it takes advantage of the number of processors it receives. For example, during a fully parallel phase, the speed-up function increases linearly with the number of processors received.

Surprisingly, in this setting, even if the scheduler is unaware of the characteristics of the phases, some policies achieve constant factor approximation of the optimal flowtime. More precisely, in [3], the author shows that the **Equi** policy, introduced in the 1980's by [12] and implemented in a lot of real systems (by time-multiplexing), achieves a competitive ratio of $(2+\sqrt{3})$ for overall completion time minimization when all the jobs arrive at time 0. [2] shows that in this setting no non-clairvoyant scheduler can achieve a competitive ratio better than $\Omega(\sqrt{n})$ when jobs arrive at arbitrary time and shows that **Equi** achieves a constant factor approximation of the optimal flowtime if it receives slightly more than twice as much resources as the optimal clairvoyant schedule it is compared to.

Our Contribution. Based on the particular case of malleable jobs, it is generally assumed in the literature that "**Equi** never starves a job" since it allocates to every job the same amount of processing power. We provide an analysis of the competitiveness of **Equi** for the makespan objective (Theorem 1) which shows that this statement is at the same time true and false: false, because, as opposed to the analysis of **Equi** for the flowtime objective in [3,2] where only the fully parallel phases count, for the makespan minimization, these phases may arbitrarily delay sequential work and thus stretch the makespan by a factor as large as, but no more than, $\frac{\ln n}{\ln \ln n}$ with respect to the optimum (Proposition 1); true, because **Equi** is as fair as can be since no algorithm can achieve a better competitive ratio up to a constant factor (Proposition 2). Furthermore, experiments in Section 5 tend to show that this worst case result may also be the typical behavior of **Equi** on random instances.

Non-clairvoyant competitiveness has been shown to be a powerful tool to analyze online strategies in various domains (for instance, [4,11]). In this paper, we aim to extend the non-clairvoyance toolbox by extending the results in [2] and [11] to the batch scheduling of sets of jobs, that go through *arbitrary* phases: each user sends, all at once, at time 0, a request for the execution of a set of jobs and is served when the last job completes. We prove that the natural algorithm **Equi**$\circ$**Equi** achieves a competitive ratio of $(2+\sqrt{3}+o(1))\frac{\ln n}{\ln \ln n}$ (Theorem 2), where n is the maximum size of a set, which is optimal up to a constant factor. We provide experimental evidences (Section 5) that this algorithm may have

the same asymptotic competitive ratio $\Theta(\frac{\ln n}{\ln \ln n})$ (independent of the number of requests) for the flowtime objective when requests have release dates, if it is given sufficiently large extra processing power with respect to the optimum.

Besides its theoretical interest, this setting corresponds to the case where several users send sets of unrelated calculations to a cluster farm that proceeds the user requests by batches, for instance because a full hardware check is performed between each pair of consecutive batches. As a byproduct of our analysis, we extend the reduction shown by Edmonds in [2, Lemma 1], by showing (Theorem 3) that one only needs to consider jobs consisting of sequential or parallel work *whatever* the objective function is (flowtime, makespan, overall set completion time, stretch, energy consumption,...) in order to treat the very wide range of non-decreasing sublinear speed-up functions all at once.

2 Non-clairvoyant Batch Sets Scheduling

The Model. We consider a collection $S = \{S_1, \ldots, S_m\}$ of sets $S_i = \{J_{i,1}, \ldots, J_{i,n_i}\}$ of n_i jobs, each of them arriving at time zero. A *schedule* $\mathcal{S}_p$ on p processors is a set of piecewise constant functions[1] $\rho_{ij} : t \mapsto \rho_{ij}^t$ where ρ_{ij}^t is the *amount of processors* allotted to job J_{ij} at time t; (ρ_{ij}^t) are arbitrary non-negative real numbers, such that at any time: $\sum_{i,j} \rho_{ij}^t \leqslant p$. Following the definition introduced by [3], each job J_{ij} goes through a series of *phases* $J_{ij}^1, \ldots, J_{ij}^{q_{ij}}$ with different degree of parallelism; the amount of *work* in each phase J_{ij}^k is w_{ij}^k; at time t, during its k-th phase, job J_{ij} progresses at a *rate* given by a *speed-up function* $\Gamma_{ij}^k(\rho_{ij}^t)$ of the amount ρ_{ij}^t of processors allotted to J_{ij}, that is to say that the amount of work accomplished between t and $t + dt$ during phase J_{ij}^k is $\Gamma_{ij}^k(\rho_{ij}^t)dt$. Let t_{ij}^k denote the completion time of the k-th phase of J_{ij}, *i.e.* t_{ij}^k is the first time t' such that $\int_{t_{ij}^{k-1}}^{t'} \Gamma_{ij}^k(\rho_{ij}^t)\, dt = w_{ij}^k$ (with $t_{ij}^0 = 0$). Job J_{ij} is completed at time $c_{ij} = t_{ij}^{q_{ij}}$. A schedule is *valid* if all jobs eventually complete, *i.e.*, $c_{ij} < \infty$ for all i, j. Set S_i is completed at time $c_i = \max_{j=1..n_i} c_{ij}$.

The Problem. The *overall completion time* of the jobs in a schedule $\mathcal{S}_p$ is: $\mathsf{CompletionTime}(\mathcal{S}_p) = \sum_{i,j} c_{ij}$. The *makespan* of the jobs in $\mathcal{S}_p$ is: $\mathsf{Makespan}(\mathcal{S}_p) = \max_{i,j} c_{ij}$. The *overall set completion time* of the sets in $\mathcal{S}_p$ is: $\mathsf{SetCT}(\mathcal{S}_p) = \sum_{i=1}^{m} c_i$. Note that: if the input collection S consists of a single set S_1, the overall set completion time of a schedule $\mathcal{S}_p$ is simply the makespan of the jobs in S_1; and if S is a collection of singleton sets $S_i = \{J_{i\,1}\}$, the overall set completion time of $\mathcal{S}_p$ is simply the overall completion time of the jobs. The overall set completion time allows then to measure a continuous range of objective functions from makespan to overall completion time. Our goal is to minimize the overall set completion time of a collection of sets of jobs arriving at time 0.

[1] Requiring the functions (ρ_{ij}) to be piecewise constant is not restrictive since any finite set of reasonable (*i.e.*, Riemann integrable) functions can be uniformly approximated from below within an arbitrary precision by piecewise constant functions. In particular, all of our results hold if ρ_{ij} are piecewise continuous functions.

We denote by $\mathrm{OPT}_p(S)$ (or simply OPT_p or OPT if the context is clear) the optimal overall set completion time of a valid schedule on p processors for collection S: $\mathrm{OPT}_p = \inf_{\text{all schedules } \mathcal{S}_p} \mathsf{SetCT}(\mathcal{S}_p)$.

Speed-Up Functions. As in [2], we make the following reasonable assumptions on the speed-up functions. In the following, we consider that each speed-up function is *non-decreasing* and *sub-linear* (*i.e.*, such that for all i, j, k, $\rho < \rho' \Rightarrow \frac{\Gamma_{ij}^k(\rho)}{\rho} \geqslant \frac{\Gamma_{ij}^k(\rho')}{\rho'}$). These assumptions are usually verified (at least desirable...) in practice: non-decreasing means that giving more processors cannot deteriorate the performances; sub-linear means that a job makes a better use of fewer processors: this is typically true when parallelism does not take too much advantage of local caches. As shown in [2], two types of speed-up functions will be of particular interest here:

- the *sequential* phase, where $\Gamma(\rho) = 1$ for all $\rho \geqslant 0$ (the job progresses at constant speed even if *no* processor is allotted to it, similarly to an idle period); and
- the *fully parallel* phase, where $\Gamma(\rho) = \rho$ for all $\rho \geqslant 0$.

Two classes of instances will be useful in the following. We denote by $(\mathsf{Par\text{-}Seq})^*$ the class of all instances in which each phase of each job is either sequential or fully parallel, and by $\mathsf{Par\text{-}Seq}$ the class of all instances in which each job consists of a fully parallel phase followed by a sequential phase. Given a $(\mathsf{Par\text{-}Seq})^*$ job J, we denote by $\mathsf{par}(J)$ (resp., $\mathsf{seq}(J)$) the sum of the fully parallel (resp., sequential) works over all the phases of J. Given a set $S_i = \{J_{i,1}, \ldots, J_{i,n_i}\}$ of $(\mathsf{Par\text{-}Seq})^*$ jobs, we denote by $\mathsf{par}(S_i) = \sum_{j=1}^{n} \mathsf{par}(J_{ij})$ and $\mathsf{seq}(S_i) = \max_{j=1,\ldots,n_i} \mathsf{seq}(J_{ij})$.

Non-clairvoyant Scheduling. As in [3,2], we consider that the scheduler knows nothing about the progress of each job and is only informed that a job is completed *at the time of its completion*. In particular, it is not aware of the different phases that the job goes through (neither of the amount of work nor of the speed-up function). It follows that even if all the job sets arrive at time 0, the scheduler has to design an *online strategy* to adapt its allocation on-the-fly to the overall progress of the jobs.

We say that a given scheduler A_p is *c-competitive* if it computes a schedule $A_p(S)$ whose overall set completion time is at most c times the optimal *clairvoyant* overall set completion time (that is aware of the characteristics of the phases of each job), *i.e.*, such that $\mathsf{SetCT}(A_p(S)) \leqslant c \cdot \mathrm{OPT}_p(S)$ for all instances S. Due to the overwhelming advantage granted to the optimum which knows all the hidden characteristics of the jobs, it is sometimes necessary for obtaining relevant informations on an non-clairvoyant algorithm to limit the power of the optimum by reducing its resources. We say that a scheduler A_p is *s-speed c-competitive* if it computes a schedule $A_{sp}(S)$ on sp processors whose overall set completion time is at most c times the optimal overall set completion time on only p processors, *i.e.*, such that $\mathsf{SetCT}(A_{sp}(S)) \leqslant c \cdot \mathrm{OPT}_p(S)$ for all instances S.

We analyse two non-clairvoyant schedulers, namely **Equi** and **Equi** $\circ$ **Equi**, and show that they have an optimal competitive ratio up to constant multiplicative

factors. The following two theorems are our main results and are proved in Propositions 1, 2 and 3.

Theorem 1 (Makespan minimization). Equi *is a* $\frac{(1+o(1))\ln n}{\ln\ln n}$*-competitive non-clairvoyant algorithm for the makespan minimization of a set of* n *jobs arriving at time* $t = 0$*. Furthermore, no non-clairvoyant deterministic (resp. randomized) algorithm is* s*-speed* c*-competitive for any* $s = o(\frac{\ln n}{\ln\ln n})$ *and* $c < \frac{\ln n}{2\ln\ln n}$ *(resp.* $c < \frac{\ln n}{4\ln\ln n}$*).*

Theorem 2 (Main result). Equi∘Equi *is a* $\frac{(2+\sqrt{3}+o(1))\ln n}{\ln\ln n}$*-competitive non-clairvoyant algorithm for the overall set completion time minimization of a collection of sets of jobs arriving at time* $t = 0$*, where* n *is the maximum cardinality of the sets. (Clearly the lower bound on competitive ratio given above holds as well for this problem).*

3 Reduction to (Par-Seq)* Instances

In [2], Edmonds shows that for the flowtime objective function, one can reduce the analysis of the competitiveness of non-clairvoyants algorithm to the instances composed of a sequence of infinitesimal sequential or parallel work. It turns out that as shown in Theorem 3 below, his reduction is far more general and applies to *any* reasonable objective function (including makespan, overall set completion time, stretch, energy consumption,...), and furthermore reduces the analysis to instances where jobs are composed of a finite sequence of positive sequential or fully parallel work, *i.e.*, to *true* (Par-Seq)* instances. It follows that for *any* non-clairvoyant scheduling problem, it is enough to analyse the competitiveness of a non-clairvoyant algorithm on (Par-Seq)* instances. Sequential and parallel phases are both unrealistic in practise (sequential phases that progress at a constant rate even if they receive no processors are not less legitimate than fully parallel phases which do not exist for real either). Nevertheless, these are much easier to handle in competitive analysis.

Consider a collection of n jobs $J_1, \ldots, J_n$ where J_i consists of a sequence of phases $J_i^1, \ldots, J_i^{q_i}$ of work $w_i^1, \ldots, w_i^{q_i}$ with speed-up functions $\Gamma_i^1, \ldots, \Gamma_i^{q_i}$. Consider a speed $s > 0$. Let A_{sp} be an arbitrary non-clairvoyant scheduler on sp processors, and $\mathcal{O}_p$ a valid schedule of $J_1, \ldots, J_n$ on p processors. The principle, see [2], is to remap the phases of the jobs within the two schedules $A_{sp}(J)$ and $\mathcal{O}_p$ as follows: each time A_{sp} allots more processors to some phase of a job than $\mathcal{O}_p$, this phase is substituted by a sequential phase and thus A_{sp} allots these resources pointlessly; and reciprocally, each time A_{sp} allots less processors to some other phase of a job than $\mathcal{O}_p$, we substitute this phase by a parallel phase. We adjust the substituted sequential and parallel works so that they fit exactly in the schedule computed by A_{sp}, which implies, as A_{sp} is non-clairvoyant, that A_{sp} will compute the exact same schedule as before; and since the instance is made easier to $\mathcal{O}_p$, the optimum can only decrease. This ensures that the competitive ratio of A_{sp} on any instance is upper bounded by the competitive ratio on (Par-Seq)* jobs.

Note that [2] implicitly assumed that the algorithm is monotonic (*i.e.*, its flowtime increases if some phase gets more work, which is the case of **Equi**), while the present reduction to (Par-Seq)* instances applies to any algorithm and furthermore to settings with release dates, precedences constraints, or any other type of constraints, since Lemma 1 simply consists in remapping the phases of the jobs *within* two *valid* schedules that already satisfy these additional constraints.

Lemma 1 (Reduction to (Par-Seq)* instances). *There exists a collection of (Par-Seq)* jobs $J'_1, \dots, J'_n$ such that $\mathcal{O}_p[J'/J]$ is a valid schedule of $J'_1, \dots, J'_n$ and $A_{sp}(J') = A_{sp}(J)[J'/J]$, where $\mathcal{S}[J'/J]$ denotes the schedule obtained by scheduling job J'_i instead of J_i in a schedule $\mathcal{S}$.*

Proof. The present proof only simplifies the proof originally given in [2] in the following ways: the jobs $J'_1, \dots, J'_n$ consist of a *finite* number of phases (and are thus a valid finitely described instance), and the schedules computed by algorithm A_{sp} on instances $J'_1, \dots, J'_n$ and $J_1, \dots, J_n$ are identical, which avoids to consider infinitely many schedules to construct J' from J.

Consider the two schedules $A_{sp}(J)$ and $\mathcal{O}_p$. Consider job J_1 (the construction of J'_i is identical for J_i, $i \geqslant 2$). Let $\rho_A(t)$ and $\rho_{\mathcal{O}}(t)$ be the number of processors allotted over time to J_1 by $A_{sp}(J)$ and $\mathcal{O}_p$ respectively. Let $\varphi(t)$ be the time t' at which the portion of work of J_1 executed in $\mathcal{O}_p$ at time t, is executed in $A_{sp}(J)$. Let $\Gamma_{t'}$ be the speed-up function of the portion of work of J_1 executed in $A_{sp}(J)$ at time t'. By construction, for all t, the same portion of work dw of J_1 is executed between t and $t+dt$ in $\mathcal{O}_p$ and between $\varphi(t)$ and $\varphi(t+dt) = \varphi(t)+d\varphi(t)$ in $A_{sp}(J)$ with the same speed-up function $\Gamma_{\varphi(t)}$, thus: $dw = \Gamma_{\varphi(t)}(\rho_{\mathcal{O}}(t))\,dt = \Gamma_{\varphi(t)}(\rho_A(\varphi(t)))\,d\varphi(t)$; it follows that φ's derivative is $\varphi'(t) = \frac{\Gamma_{\varphi(t)}(\rho_{\mathcal{O}}(t))}{\Gamma_{\varphi(t)}(\rho_A(\varphi(t)))}$ ($\geqslant 0$, φ is an increasing function). $\rho_A(\varphi(t))$ and $\rho_{\mathcal{O}}(t)$ are (by definition) piecewise constant functions. Let $t_1 = 0 < t_2 < \cdots < t_\ell$ such that $\rho_A(\varphi(t))$ and $\rho_{\mathcal{O}}(t)$ are constant on each time interval $[t_k, t_{k+1})$ and zero beyond t_ℓ; let $t'_k = \varphi(t_k)$, $\rho_A(t')$ is constant on each time interval (t'_k, t'_{k+1}); let $\rho^k_A = \rho_A(t'_k)$ and $\rho^k_{\mathcal{O}} = \rho_{\mathcal{O}}(t_k)$. By construction, the portion of work of J_1 executed by $A_{sp}(J)$ between times t'_k and t'_{k+1}, is executed by $\mathcal{O}_p$ between times t_k and t_{k+1}. J'_1 consists of a sequence of $(\ell-1)$ phases, sequential or fully parallel depending on the relative amount of processors $\rho^k_{\mathcal{O}}$ and ρ^k_A alloted by $\mathcal{O}_p$ and $A_{sp}(J)$ to J_1 during time intervals $[t_k, t_{k+1}]$ and $[t'_k, t'_{k+1}]$ respectively. The k-th phase of J'_1 is defined as follows:

- If $\rho^k_{\mathcal{O}} \leqslant \rho^k_A$, the k-th phase of J'_1 is a sequential work of $w_k = t'_{k+1} - t'_k$.
- If $\rho^k_{\mathcal{O}} > \rho^k_A$, the k-th phase of J'_1 is a fully parallel work of $w_k = \rho^k_A \cdot (t'_{k+1} - t'_k)$.

The k-th phase of J'_1 is designed to fit exactly in the overall amount of processors allotted by A_{sp} to J_1 during $[t'_k, t'_{k+1}]$; thus, since A_{sp} is non-clairvoyant, $A_{sp}(J') = A_{sp}(J)[J'/J]$. Let now verify that the k-th phase of J'_1 fits in the overall amount of processors allotted by $\mathcal{O}_p$ to J_1 during $[t_k, t_{k+1}]$.

- If $\rho^k_{\mathcal{O}} \leqslant \rho^k_A$, $w_k = \int_{t'_k}^{t'_{k+1}} dt' = \int_{t_k}^{t_{k+1}} \varphi'(t)dt = \int_{t_k}^{t_{k+1}} \frac{\Gamma_{\varphi(t)}(\rho^k_{\mathcal{O}})}{\Gamma_{\varphi(t)}(\rho^k_A)} dt \leqslant \int_{t_k}^{t_{k+1}} dt = t_{k+1} - t_k$ since the $\Gamma_{\varphi(t)}$ are non-decreasing functions.

– If $\rho^k_{\mathcal{O}} > \rho^k_A$, $w_k = \rho^k_A \int_{t'_k}^{t'_{k+1}} dt' = \rho^k_A \int_{t_k}^{t_{k+1}} \frac{\Gamma_{\varphi(t)}(\rho^k_{\mathcal{O}})}{\Gamma_{\varphi(t)}(\rho^k_A)} dt \leqslant \rho^k_A \int_{t_k}^{t_{k+1}} \frac{\rho^k_{\mathcal{O}}}{\rho^k_A} dt = \rho^k_{\mathcal{O}} \cdot (t_{k+1} - t_k)$, since the $\Gamma_{\varphi(t)}$ are sub-linear functions.

It follows that in both cases, the k-th phase of J'_1 can be completed in the space allotted to J_1 in $\mathcal{O}_p$ during $[t_k, t_{k+1}]$. □

Consider an arbitrary non-clairvoyant scheduling problem where the goal is to minimize an objective function F over the set of all valid schedules of an instance of jobs $J_1, \ldots, J_n$. Assume that F is *monotonic* in the sense that if $\mathcal{S}$ and $\mathcal{S}[J'/J]$ are valid schedules of J and J' respectively, then $F(\mathcal{S}[J'/J]) \leqslant F(\mathcal{S})$, which means essentially that wasting resources (allocating more resources than needed or allocating processors after the completion of a job) costs no more than using them: putting a larger thing into a box costs at least as much as putting a smaller thing into the same box. Note that *all* standard objective functions are monotonic: flowtime, makespan, overall completion time, overall set completion time, stretch, energy consumption, etc. Then,

Theorem 3. *Any non-clairvoyant algorithm A^F for a monotonic objective function F that is s-speed c-competitive over* (Par-Seq)* *instances, is also s-speed c-competitive over all instances of jobs going through phases with arbitrary non-decreasing sublinear speed-up functions.*

Proof. Consider a non-(Par-Seq)* instance $J = \{J_1, \ldots, J_n\}$. Denote by $\mathrm{OPT}^F_p(J)$ the optimal cost for J, *i.e.*, $\mathrm{OPT}^F_p(J) = \inf\{F(\mathcal{S}) : \mathcal{S}$ is a valid schedule of J on p processors$\}$. Consider an arbitrary small $\epsilon > 0$ and $\mathcal{O}$ a valid schedule of J such that $F(\mathcal{O}) \leqslant \mathrm{OPT}^F_p(J) + \epsilon$ (note that we do not need that an optimal schedule exists). Let J' be the (Par-Seq)* instance given by Lemma 1 from J, A^F_{sp}, and $\mathcal{O}$. Since $A^F_{sp}(J') = A^F_{sp}(J)[J'/J]$, $F(A^F_{sp}(J)) = F(A^F_{sp}(J'))$. But A^F_{sp} is s-speed c-competitive for J', so: $F(A^F_{sp}(J)) \leqslant c \cdot \mathrm{OPT}^F_p(J') \leqslant c \cdot F(\mathcal{O}[J'/J]) \leqslant c \cdot F(\mathcal{O}) \leqslant c\,\mathrm{OPT}^F_p(J) + c\,\epsilon$, as $\mathcal{O}[J'/J]$ is a valid schedule of J' and F is monotonic. Decreasing ϵ to zero completes the proof. □

We shall from now on consider only (Par-Seq)* instances.

4 Fairness Is Fair Enough

4.1 The Single Set Case

In this section, we focus on the case where the collection S consists of a unique set $S_1 = \{J_1, \ldots, J_n\}$. The problem consists thus in minimizing the *makespan* of the set of jobs S_1. This problem is interesting on its own and, as far as we know, no competitive non-clairvoyant algorithm was known. Furthermore, the analysis that follows is one of the keys to the main result of the next section.

Equi *Algorithm.* **Equi** is the classic operating system approach to non-clairvoyant scheduling. It consists in giving an equal amount of processors to each uncompleted job (operating systems approximate this strategy by a pre-emptive round robin policy). Formally, given p processors, if $N(t)$ denotes the number of uncompleted jobs at time t, **Equi** allots $\rho_i^t = p/N(t)$ processors to each uncompleted job J_i at time t.

Analysis of Equi for makespan minimization. Thanks to Theorem 3, we focus on a $(\mathsf{Par\text{-}Seq})^*$ instance $S = \{J_1, \ldots, J_m\}$. By rescaling the parallel work in each job, we can assume w.l.o.g. that $p = 1$. Let us define the Par-Seq instance $S' = \{J'_1, ..., J'_n\}$ where each J'_i consists of a fully parallel phase of work $\mathsf{par}(J_i)$ followed by a sequential phase of work $\mathsf{seq}(J_i)$. Observe that:

Lemma 2. $\mathsf{Makespan}(\mathbf{Equi}(S')) \geqslant \mathsf{Makespan}(\mathbf{Equi}(S))$.

Proof. Since all the jobs arrive at time 0, the number of uncompleted jobs is a non-increasing function of time. It follows that the amount of processors alloted by **Equi** to a given job is a non-decreasing function of time. Thus, moving all the parallel work to the front, can only delay the completion of the jobs since less processors will then be allocated to each given piece of parallel work. □

Now, every job consists of a parallel phase followed by a sequential phase of length at most $\mathsf{seq}(S')$. The key to the analysis is to observe that: if more than a proportion α of jobs are in a parallel phase, then the overall parallel work progresses at a rate at least α; and if more than a proportion $(1-\alpha)$ of jobs are in a sequential phase then after $\mathsf{seq}(S')$ time, these jobs are completed and the number of jobs decrease by a factor α. It follows that the parallel work is at most dilated by some factor $1/\alpha$ and the sequential phases get extended by some logarithmic factor. We thus obtain the following competitive ratio:

Proposition 1. Equi *is* $(1 + o(1))\frac{\ln n}{\ln \ln n}$*-competitive for the makespan minimization problem.*

Proof. Consider the schedule $\mathbf{Equi}(S')$ and let $T = \mathsf{Makespan}(\mathbf{Equi}(S'))$. We write $[0, T]$ as the disjoint union of two sets A and $\bar{A}$. Set $\alpha = \frac{(\ln \ln n)^2}{\ln n}$. Recall that $N(t)$ is the number of uncompleted jobs at time t. Let s_t be the number of uncompleted jobs in a sequential phase at time t. Set A is the set of all the instants where the fraction of jobs in a sequential phase is larger than α, and $\bar{A}$ is its complementary set: *i.e.*, $A = \{0 \leqslant t \leqslant T : s_t \geqslant (1-\alpha)N(t)\}$ and $\bar{A} = \{0 \leqslant t \leqslant T : s_t < (1-\alpha)N(t)\}$. Clearly, $T = |A| + |\bar{A}|$, with $|X| = \int_X dt$. We now bound $|A|$ and $|\bar{A}|$ independently.

At any time t in $\bar{A}$, the total amount of parallel work completed between t and $t + dt$ is at least $\alpha\, dt$. Since the total amount of parallel work is $\mathsf{par}(S')$, we get $\int_{\bar{A}} \alpha\, dt \leqslant \mathsf{par}(S')$. Thus, $|\bar{A}| \leqslant \mathsf{par}(S')/\alpha$.

Now, let $t_1 < \cdots < t_q$ with $t_k \in A$ for all k, such that the time intervals $I_1 = [t_1, t_1 + \mathsf{seq}(S')), \ldots, I_q = [t_q, t_q + \mathsf{seq}(S'))$ form a collection of non-overlapping intervals of length $\mathsf{seq}(S')$ that covers A. Once the sequential phase of a Par-Seq job has begun at or before time t, the job

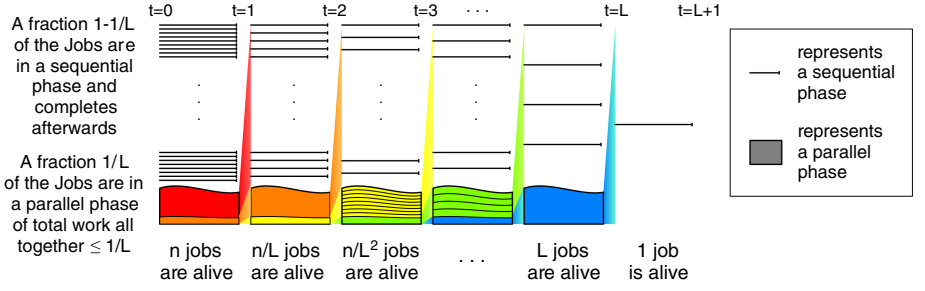


Fig. 1. Worst case instance designed by the mean adversary

completes before time $t + \mathsf{seq}(S')$. Since at time t_k, at least $(1-\alpha)\cdot N(t_k)$ jobs are in a sequential phase, at time $t_{k+1} \geqslant t_k + \mathsf{seq}(S')$, we have thus: $N(t_{k+1}) \leqslant \alpha N(t_k)$. It follows that $N(t_k) \leqslant \alpha^k \cdot n$. Since $N(t_q) \geqslant 1$, $q \leqslant \frac{\ln n}{\ln(1/\alpha)}$. But A is covered by q time intervals of length $\mathsf{seq}(S')$, so: $|A| \leqslant \frac{\ln n}{\ln(1/\alpha)}\,\mathsf{seq}(S')$. Finally, $\mathsf{Makespan}(\mathbf{Equi}(S)) \leqslant \mathsf{Makespan}(\mathbf{Equi}(S')) = T \leqslant \frac{1}{\alpha}\,\mathsf{par}(S') + \frac{\ln n}{\ln(1/\alpha)}\,\mathsf{seq}(S') \leqslant (1+o(1))\frac{\ln n}{\ln\ln n}\max(\mathsf{par}(S'),\mathsf{seq}(S')) = (1+o(1))\frac{\ln n}{\ln\ln n}\max(\mathsf{par}(S),\mathsf{seq}(S)) \leqslant (1+o(1))\frac{\ln n}{\ln\ln n}\mathrm{OPT}(S)$. □

Equi is asymptotically optimal up to a factor 2. The following lemma shows that **Equi** is asymptotically optimal in the worst case. Note that increasing the number of processors by a factor s does *not* improve the competitive ratio of any deterministic or randomized algorithm as long as $s = o(\frac{\ln n}{\ln\ln n})$, *i.e.*, the competitive ratio does not improve even if the number of processors increases (not too fast) with the number of jobs. Figure 1 presents the worst case instance used below.

Proposition 2 (Lower bound for any non-clairvoyant algorithm). *No non-clairvoyant algorithm A has a competitive ratio less than $\gamma_D = \frac{\ln n}{2\ln\ln n}$ if A is deterministic, and $\gamma_R = \frac{\ln n}{4\ln\ln n}$ if A is randomized.*

Furthermore, no non-clairvoyant algorithm A is s-speed c-competitive for any speed $s = o(\frac{\ln}{\ln\ln n})$ if $c < \gamma_D$ and A is deterministic, or $c < \gamma_R$ if A is randomized.

Proof. Consider the execution of an algorithm A_s given s processors on the following instance (see Fig. 1). At time 0, $n = (s\ell)^\ell$ jobs are given. Since the algorithm is non-clairvoyant, we set the phase afterwards. At time 1, we renumber the jobs $J_1, \ldots, J_n$ by non-decreasing processing power received between $t = 0$ and $t = 1$ in A_s. Between time $t = 0$ and $t = 1$, we set the jobs $J_{(s\ell)^{\ell-1}+1}, \ldots, J_n$ (*i.e.*, the last fraction $1 - 1/(s\ell)$ of the $(s\ell)^\ell$ jobs) to be in a sequential phase of work 1 and say that all of them complete at time 1; each J_j of the $J_1, \ldots, J_{(s\ell)^{\ell-1}}$ are set in a parallel phase of work $\int_0^1 \rho_j^t\, dt$ each between time 0 and 1, where ρ_j^t is the amount of processors alloted to J_i at time t. The processing power received by the last $1 - 1/(s\ell)$ fraction of jobs between $t = 0$ and $t = 1$ is at least $s - 1/\ell$

and thus, the total parallel work assigned to the jobs between 0 and 1 is at most $1/\ell$. At time 1 only remains the jobs $J_1, \ldots, J_{(s\ell)^{\ell-1}}$ that just have finished their first parallel phase. We continue recursively as follows until time $t = \ell$: at integer time $t = i < \ell$, $(s\ell)^{\ell-i}$ jobs are still uncompleted; between time $t = i$ and $t = i+1$, the fraction $1 - 1/(s\ell)$ of the $(s\ell)^{\ell-i}$ jobs that received the most processing power are set in a sequential phase of work 1 and all of them complete at time $i+1$; each job J_j of the other $1/(s\ell)$ fraction is set in a parallel phase of work $\int_i^{i+1} \rho_j^t \, dt$ each; At time $i+1$ only remains the $(s\ell)^{\ell-i}/(s\ell) = (s\ell)^{\ell-(i+1)}$ jobs that just have finished their i-th parallel phase. At time $t = \ell$, there only remains one job which completes at time $\ell+1$ after a sequential phase of work 1. It follows that for this instance, A_s achieves a makespan of $\ell+1$. But, the amount of parallel work executed within each time interval $[i, i+1]$ for $i = 0, ..., \ell-1$, is at most $1/\ell$. It follows that an optimal (clairvoyant) scheduler on 1 processor can complete all the parallel work in one time unit and then finish the remaining sequential work before time 2. But $n = (s\ell)^\ell$, $\ell > \frac{\ln n}{\ln \ln n}$, which concludes the proof. *(The proof for randomized algorithms, based on Yao's principle [13,10], is omitted due to space constraints).* □

4.2 Non-clairvoyant Batch Set Scheduling

We now go back to the general problem. Consider a collection $S = \{S_1, \ldots, S_m\}$ of m sets $S_i = \{J_{i,1}, \ldots, J_{i,n_i}\}$ of n_i $(\mathsf{Par\text{-}Seq})^*$ jobs, each of them arriving at time zero. The goal is to minimize the overall set completion time of the sets.

Equi∘Equi *Algorithm.* We consider the **Equi∘Equi** strategy which splits evenly the amount of processors given to each set among the uncompleted jobs within that set. Formally, let $N(t)$ be the number of uncompleted sets at time t, and $N_i(t)$ the number of uncompleted jobs in each uncompleted set S_i at time t. At time t, **Equi∘Equi** on p processors allots to each uncompleted job J_{ij} an amount of processors $\rho_{ij}^t = \frac{p}{N(t) \cdot N_i(t)}$. The following section shows that indeed the competitive ratio of this strategy is asymptotically optimal (up to a constant multiplicative factor).

Competitiveness of **Equi∘Equi**. Scaling by a factor p each sequential work, again we assume w.l.o.g. that $p = 1$. Consider the $\mathsf{Par\text{-}Seq}$ instance $S' = \{S'_1, \ldots, S'_m\}$ where $S'_i = \{J'_{i,1}, \ldots, J'_{i,n_i}\}$ and each job J'_{ij} consists of a fully parallel phase of work $\mathsf{par}(J_{ij})$ followed by a sequential phase of work $\mathsf{seq}(J_{ij})$. Following the proof of Lemma 2, we get:

Lemma 3. $\mathsf{SetCT}(\mathbf{Equi} \circ \mathbf{Equi}(S')) \geqslant \mathsf{SetCT}(\mathbf{Equi} \circ \mathbf{Equi}(S))$.

The next lemmas are the keys to the result. They reduce the analysis of **Equi∘Equi** to the analysis of the *overall completion time* of **Equi** for a collection of *jobs*, which is known from [3] to be $(2+\sqrt{3})$-competitive when all the jobs arrive at time 0. Let $n = \max_{i=1,\ldots,m} n_i$ be the maximum size of a set S_i, and let $\alpha = \frac{(\ln \ln n)^2}{\ln n}$. The principle is to reduce the analysis of sets of jobs to the analysis of single $(\mathsf{Par\text{-}Seq})^*$ jobs as follows. Consider the lifetime of a set, each

time a proportion more than α of the jobs are in a parallel phase within the set, we create a parallel phase; and each time a proportion more than $(1-\alpha)$ of the jobs are in sequential phase, we create a sequential phase. For the same raison as before, the overall parallel work gets dilated by at most $1/\alpha$ while the overall sequential work is extended by at most a logarithmic factor.

Lemma 4. *There exists a* (Par-Seq)* *instance* $J = \{J_1, \ldots, J_m\}$ *of Non-Clairvoyant Batch* Job *Scheduling, such that:* $\mathbf{Equi}(J) = \mathbf{Equi} \circ \mathbf{Equi}(S')[J/S']$, $\mathsf{par}(J_i) \leqslant \frac{1}{\alpha}\mathsf{par}(S'_i)$, *and* $\mathsf{seq}(J_i) \leqslant \frac{\ln n}{\ln(1/\alpha)}\mathsf{seq}(S'_i)$, *where* $\mathcal{S}[J/S']$ *denotes the schedule where* J_i *receives at any time the total amount of processors alloted to the jobs* J'_{ij} *of* S'_i *in schedule* $\mathcal{S}$.

Proof. Let $\mathcal{E} = \mathbf{Equi} \circ \mathbf{Equi}(S')$. Let us construct J_1 (the construction of J_i, $i \geqslant 2$, is identical). Consider the jobs $J'_{1,1}, \ldots, J'_{1,n_1}$ of S'_1 in the schedule $\mathcal{E}$. Let $t_1 = 0 < \cdots < t_q = c'_1$ (where c'_1 denotes the completion time of S'_1 in $\mathcal{E}$), such that during each time interval $[t_k, t_{k+1})$, each job $J'_{1,j}$ remains in the same phase; during $[t_k, t_{k+1})$, the number of jobs of S'_1 in a sequential (resp. fully parallel) phase is constant, say s_k (resp. $N_1(t_k) - s_k$). J_1 has $(q-1)$ phases:

- if $s_k \geqslant (1-\alpha)N_1(t_k)$, the k-th phase of J_1 is sequential of work $w_k = t_{k+1} - t_k$.
- if $s_k < (1-\alpha)N_1(t_k)$, the k-th phase of J_1 is fully parallel of work $w_k = \int_{t_k}^{t_{k+1}} \frac{1}{N(t)} dt$.

J_1 is designed to fit exactly in the space alloted to S'_1 in $\mathcal{E}$, thus $\mathbf{Equi}(J) = \mathcal{E}[J/S']$. We now have to bound the total parallel and total sequential works in J_1. Let $K = \{k : s_k \geqslant (1-\alpha)N_1(t_k)\}$ and $\bar{K} = \{1, \ldots, q-1\} \setminus K$; by construction, $\mathsf{seq}(J_1) = \sum_{k \in K} w_k$ and $\mathsf{par}(J_1) = \sum_{k \in \bar{K}} w_k$. For each $t \in [t_k, t_{k+1})$ with $k \in \bar{K}$, the amount of parallel work of jobs in S'_1 between t and $t + dt$ is at least $\frac{\alpha N_1(t)}{N(t) \cdot N_1(t)}\, dt = \frac{\alpha}{N(t)}\, dt$. It follows that the amount of parallel work of jobs in S'_1 scheduled in $\mathcal{E}$ during $[t_k, t_{k+1})$ is at least $\alpha \int_{t_k}^{t_{k+1}} \frac{1}{N(t)} dt = \alpha\, w_k$. Thus, $\mathsf{par}(S'_1) \geqslant \sum_{k \in \bar{K}} \alpha\, w_k = \alpha\, \mathsf{par}(J_1)$, which is the claimed bound. Now, let $A = \cup_{k \in K}[t_k, t_{k+1})$, we have $|A| = \mathsf{seq}(J_1)$. Since the bound on the size of A in proof of Proposition 1 relies on a counting argument (and is thus independent of the amount of processors given to the set) and the jobs in S'_1 are Par-Seq, the same argument applies and $|A| \leqslant \frac{\ln n_1}{\ln(1/\alpha)}\mathsf{seq}(S'_1) \leqslant \frac{\ln n}{\ln(1/\alpha)}\mathsf{seq}(S'_1)$, which conclude the proof. □

Let $J' = \{J'_1, \ldots, J'_m\}$ be the Par-Seq instance of Batch Job Scheduling where each job J'_i consists of a fully parallel work of $\mathsf{par}(J_i)$ followed by a sequential work of $\mathsf{seq}(J_i)$. Again, as the amount of processors alloted by **Equi** to each job is a non-decreasing function of time, pushing parallel work upfront can only make it worse, thus: $\mathbf{Equi}(J) \leqslant \mathbf{Equi}(J')$.

We can now conclude on the competitiveness of $\mathbf{Equi} \circ \mathbf{Equi}$ since **Equi** is proved to be $(2+\sqrt{3})$-competitive for instance J' in [3, Theorem 3.1].

Proposition 3. **Equi∘Equi** *is* $\frac{(2+\sqrt{3}+o(1))\ln n}{\ln \ln n}$*-competitive for the overall set completion time minimization problem.*

5 Experimental Study of Equi and Equi∘Equi

Fig. 2(a) presents the empirical overall and maximum competitive ratio of **Equi** for the Makespan objective on random stream-lined instances [2]: for each n, a set of n jobs, of $5n$ phases of unit work each, is requested at time 0; for all i, the type of the i-th phase of the n jobs are defined as follows: a job is chosen uniformly at random and its i-th phase is set to parallel, and the i-th phases of all the other jobs are all sequential. It appears that the empirical competitive ratio measured for **Equi** on this random instance appears to be asymptotically $\sim .7\frac{\ln n}{\ln\ln n}$. The worst case ratio proven in Proposition 2 seems thus to be the typical behavior of **Equi**.

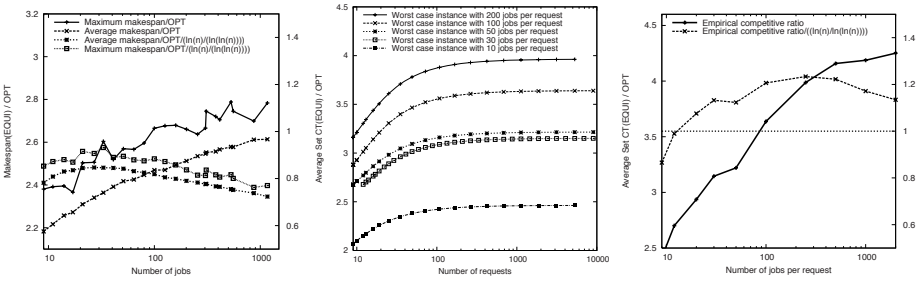


Fig. 2. From left to right: a) Makespan of **Equi** on random stream-lined instances; b) Average Set Flowtime of **Equi∘Equi** on a stream of worst case instances for Makespan on 2 processors; c) Worst empirical competitive ratio for **Equi** $\circ$ **Equi**$_2$ w.r.t. OPT_1

Fig. 2(b) and (c) present the competitive ratio of **Equi∘Equi** for the Average Set Flowtime objective (the average flowtime of each set of jobs) with 2 processors with respect to the optimal with 1 processor, on the following instance: at each time step $t = 0..10\,000$, we request an instance of the worst case type described in Fig. 1 with n jobs, $n \in \{10, 30, 50, 100, 200\}$. Fig. 2(c) shows that the empirical competitive ratio seems to tend asymptotically to $\sim \frac{\ln n}{\ln\ln n}$ as time grows. It seems thus that the competitive ratio of **Equi∘Equi** is independent of the number of requests for the Average Set Flowtime objective as well.

References

1. Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., Weglarz, J.(eds.): Handbook on Scheduling: Models and Methods for Advanced Planning, chapter Online Scheduling. International Handbooks on Information Systems. Springer, Heidelberg (2007), at http://www.cs.pitt.edu/~kirk/papers/index.html
2. Edmonds, J.: Scheduling in the dark. In: Proc. of the 31st ACM Symp. on Theory of Computing (STOC), pp. 179–188. ACM Press, New York (1999)
3. Edmonds, J., Chinn, D.D., Brecht, T., Deng, X.: Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. J. Scheduling 6(3), 231–250 (2003)

4. Edmonds, J., Pruhs, K.: Broadcast scheduling: when fairness is fine. In: Proc. of the 13th annual ACM-SIAM symposium on Discrete algorithms (SODA 2002), Philadelphia, PA, USA, pp. 421–430. ACM Press, New York (2002)
5. Feldmann, A., Kao, M.-Y., Sgall, J., Teng, S.-H.: Optimal online scheduling of parallel jobs with dependencies. J. of Combinatorial Optimization 1, 393–411 (1998)
6. Graham, R.L.: Bounds for certain multiprocessing anomalies. Bell System Technical Journal 45, 1563–1581 (1966)
7. Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM Journal on Applied Mathematics 17, 263–269 (1969)
8. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. J. ACM 47(4), 617–643 (2000)
9. Motwani, R., Philipps, S., Torng, E.: Non-clairvoyant scheduling. Theoretical Computer Science 130, 17–47 (1994)
10. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
11. Robert, J., Schabanel, N.: Pull-based data broadcast with dependencies: Be fair to users, not to items. In: Proc. of Symp. on Discrete Algorithms (SODA) (2007)
12. Tucker, A., Gupta, A.: Process control and scheduling issues for mulitprogrammed shared memory multiprocessors. In: Proc. of the 12th ACM Symp. on Op. Syst. Principles, pp. 159–166. ACM Press, New York (1989)
13. Yao, A.: Probabilistic computations: Towards a unified measure of complexity. In: Proc. of 17th Symp. on Fond. of Computer Science (FOCS), pp. 222–227 (1977)

An Experimental Study of New and Known Online Packet Buffering Algorithms*

Susanne Albers and Tobias Jacobs

University of Freiburg, Georges Köhler Allee 79, 79110 Freiburg, Germany
{salbers,jacobs}@informatik.uni-freiburg.de

Abstract. We present the first experimental study of online packet buffering algorithms for network switches. We consider a basic scenario in which m queues of size B have to be maintained so as to maximize the packet throughput. For this model various online algorithms with competitive factors ranging between 2 and 1.5 were developed in the literature. We first develop a new 2-competitive online algorithm, called *HSFOD*, which is especially designed to perform well under real-world conditions. In our experimental study we have implemented all the proposed algorithms, including *HSFOD*, and tested them on packet traces from benchmark libraries. We have evaluated the experimentally observed competitiveness, the running times, memory requirements and actual packet throughput of the strategies. The tests were executed for varying values of m and B as well as varying switch speeds. It shows that greedy-like strategies and *HSFOD* perform best in practice.

1 Introduction

Over the past five years the algorithms community has witnessed tremendous research interest in packet buffering algorithms. Given a network router or switch that is equipped with packet buffers of limited capacity, the general goal is to design strategies for serving these buffers so as to maximize the total packet throughput. While packet buffering policies have been investigated in the applied computer science and, in particular, networking communities for many years, only seminal papers by Aiello et al. [1] and Kesselman et al. [8] have initiated theoretical and algorithmic studies. These studies and those presented since aim at analyzing existing algorithms and at designing new strategies with a provably good performance.

Packet buffering is an online problem in that data packets arrive over time and, at any time, future packet arrivals are unknown. Results from queueing theory cannot be applied directly as network traffic exhibits so-called *self-similar* properties, cf. [6,14]. Therefore, algorithmic research resorts to competitive analysis [12], comparing an online algorithm A to an optimal offline algorithm OPT that knows the entire packet arrival sequence in advance. Algorithm A is called c-competitive if, for all packet arrival sequences, the throughput achieved by A

* Work supported by the German Research Foundation, projects AL 464/4-1 and 4-2.

L. Arge, M. Hoffmann, and E. Welzl (Eds.): ESA 2007, LNCS 4698, pp. 754–765, 2007.

is at least $1/c$ times that of OPT. In the above-mentioned algorithmic body of work, various packet buffering problems were investigated. The following natural questions arise: Do the competitive analyses give meaningful results? Are the proposed new algorithms interesting from a practical point of view? Does optimizing the worst-case behaviour also improve the practical performance? So far, these issues were not addressed.

In this paper we present the first experimental study of online packet buffering algorithms. We consider a scenario that is very basic and has been investigated the most among the proposed models, see [2,3,4,8,10]. Specifically, we are given m packet buffers, each of which is associated with an input port of a switch. Each buffer is organized as a queue and can simultaneously store up to B data packets. The capacity B is also referred to as the *size* of the buffer. Time is assumed to be discrete. Each time step consists of two phases, namely a *packet arrival phase* and a *packet transmission phase*. At any time, in the packet arrival phase, new packets may arrive at the buffers. Let b_i be the number of packets currently stored in buffer i, and let a_i be the number of newly arriving packets at that buffer. If $a_i + b_i \leq B$, then all new packets can be accepted; otherwise $a_i + b_i - B$ packets must be dropped. Furthermore, at any time, in the packet transmission phase, an algorithm can select one non-empty buffer and transfer the packet at the head of that queue to the output port. We assume w.l.o.g. that the packet arrival phase precedes the transmission phase. The goal is to maximize the throughput, i.e. the total number of transferred packets. We emphasize that we consider all packets to be equally important, i.e. all of them have the same value. The scenario we study here arises, for instance, in input-queued (IQ) switches which represent the dominant switch architecture today.

Known algorithms: The most simple and natural packet buffering algorithm is the *Greedy* policy: At any time serve the queue currently storing the largest number of packets. Unfortunately, *Greedy* has essentially the worst possible competitive ratio. It is easy to show [2,4] that any *work conserving* algorithm, which at any time serves an arbitrary non-empty buffer, is 2-competitive. Obviously, *Greedy* belongs to the class of work conserving strategies. It was shown in [2] that the competitive ratio of *Greedy* is not smaller than $2 - 1/B$, no matter how ties are broken. Thus *Greedy* has a competitiveness of exactly 2, for arbitrary buffer sizes. The first deterministic algorithm that achieved a competitive ratio below 2 was devised in [2]. The proposed *Semi Greedy* algorithm deviates from standard *Greedy* when the buffer occupancy is low and has a competitive performance of $17/9 \approx 1.89$. The deterministic strategy with the smallest competitive ratio known is the *Waterlevel* algorithm [3] with a competitiveness of $\frac{e}{e-1}(1 + \frac{\lfloor H_m+1 \rfloor}{B})$, where H_m is the m-th Harmonic number. This ratio is optimal for large B, as no deterministic algorithm can have a competitive ratio smaller than $e/(e-1) \approx 1.58$, see [2]. As for randomized strategies, a *Random Schedule* algorithm [4] achieves a competitive ratio of $e/(e-1)$ while a *Random Permutation* algorithm is 1.5-competitive [10]. These performance ratios hold against oblivious adversaries and are close to the best lower bound of 1.46, see [2]. The five algorithms just mentioned comprise all online strategies known in the

literature for our packet buffering problem. As for the offline problem, a polynomial time algorithm computing optimal solutions was given in [2].

Contributions of this paper: We first introduce a new online packet buffering algorithm called *HSFOD*. It is based on the idea to estimate the packet arrival rate for each port. In each time step the algorithm transmits a packet from a non-empty queue that, according to these arrival rates, encounters packet loss earliest in the future assuming buffers would not be served anymore. This new strategy is presented in Section 2. We prove that it is 2-competitive.

The major part of this paper is devoted to an extensive experimental study of the packet buffering problem under consideration. The main purpose of our experiments is to determine the experimentally observed competitiveness of all the proposed online algorithms and to establish a relative performance ranking among the strategies. As the name suggests, the experimentally observed competitiveness is the ratio of the throughput of an online algorithm to that of an optimal solution as it shows in experimental tests. Additionally, we wish to evaluate the running times and memory requirements of the algorithms as some of the strategies are quite involved and need auxiliary data structures. Finally, we are interested in the actual throughput in terms of the total number of successfully transmitted packets.

We have tested the algorithms on real-world traces. We selected traces from the Internet Traffic Archive [7], which is a moderated trace repository sponsored by ACM SIGCOMM. In our experiments we have studied varying port numbers m as well as varying buffers sizes B. Furthermore, we have investigated the influence of varying the *speed* of a switch, i.e. the frequency with which it can forward packets. We have adjusted this parameter relative to the given data traces. For instance, a speed of value 1 indicates that the *average* packet arrival frequency is equal to the frequency with which packets can be transmitted.

In Section 3 we present a concise description of the five previously known online buffering algorithms as well as the optimal offline strategy. For all the proposed strategies, including *HSFOD*, we describe how the given pseudo-code was indeed implemented and discuss runtime issues as well as extra space requirements of the strategies. We implemented the data model and the algorithms using the Java programming language. The test environment is described in Section 4. A detailed presentation of the results follows in Section 5. One of the most important findings is that the experimentally observed competitiveness is much lower than the theoretical bounds. Typically, the online algorithms are at most 3% worse than an optimal offline algorithm. In fact, *HSFOD* shows the best performance, having a gap of less and 0.1%. We remark here that *HSFOD* was designed after we had implemented and evaluated the previously known algorithms. Hence it can be viewed as a result of an algorithm engineering process. Furthermore, the theoretical competitive ratios are no proper indication of how the algorithms perform in practice. The randomized algorithms, despite their low theoretical competitiveness, do not perform better than the deterministic ones. From a practical point of view *Greedy*, *HSFOD* and *Semi Greedy* are the algorithms of choice.

2 The New Algorithm *HSFOD*

The new strategy we introduce estimates future packet arrival rates by keeping track of past arrival patterns. Based on these arrival rates, the algorithm mimics the optimal offline algorithm *SFOD* [2] which at each point of time transmits a packet from a non-empty queue that would overflow earliest in the future if no queues were served.

Algorithm HSFOD: For each input port the algorithm maintains a weighted moving average estimating the packet arrival rate. Let $r_i(t)$ be the rate at time t for port i, $1 \leq i \leq m$. Then $r_i(t) = \alpha \cdot r_i(t-1) + (1-\alpha) \cdot a_i(t)$, where $a_i(t)$ is the number of packets that have just arrived at port i, and $\alpha \in (0,1)$ is some fixed constant. We set $r_i(0) = 0$ initially. The overflow time for each port i is calculated as $t_i^{\text{ov}}(t) = (B - b_i(t))/r_i(t)$, where $b_i(t)$ is the number of packets currently stored in buffer i. Note that $r_i(t)$ and $t_i^{\text{ov}}(t)$ are allowed to take fractional values. At any time the algorithm serves the buffer that has the smallest overflow time; ties may be broken arbitrarily.

Theorem 1. *The competitive ratio of* HSFOD *is exactly equal to 2.*

The proof is given in the full paper. *HSFOD* depends on a parameter $0 < \alpha < 1$ that weights past and current packet arrivals. Experimental evaluations show that *HSFOD* achieves the best practical performance for values of α in the range $0.995 \leq \alpha \leq 0.997$. Again, details are given in the full paper. We set α to 0.997 in the experiments described in the following sections.

3 The Implemented Algorithms

We describe implementation issues for all the known strategies that we have evaluated experimentally. Due to space constraints we only consider the online algorithms here. As the optimal offline algorithm *SFOD* is just used for comparison, we omit its discussion in this extended abstract.

As we consider a scenario where all packets have the same value, unless otherwise stated, the algorithms apply a greedy admission policy: At any time t and for any of the m buffers, whenever new data packets arrive, an algorithm accepts as many packets as possible subject to the constraint that a buffer can only store up to B packets simultaneously. Thus the algorithms only specify which buffer to serve in each time step. We will use the terms *buffer* and *queue* interchangeably and use q_i to refer to the i-th buffer/queue. Let the *load* of a queue be the number of packets currently stored in it.

3.1 Deterministic Online Algorithms

Algorithm Greedy: In each time step serve the queue currently having the maximum load; ties may be broken arbitrarily.

Algorithm Semi Greedy: In each time step execute the first of the following three rules that applies to the current buffer configuration. (1) If there is a

queue buffering more than $\lfloor B/2 \rfloor$ packets, serve the queue currently having the maximum load. (2) If there is a queue the hitherto maximum load of which is less than B, then among these queues serve the one currently having the maximum load. (3) Serve the queue currently having the maximum load. Whenever all queues become empty, the hitherto maximum load is reset to 0 for all queues.

In our implementation of *Greedy* and *Semi Greedy* we use auxiliary heap data structures to determine the queues having the maximum load. Ties are broken by choosing the queue with the smallest index.

We next give a condensed presentation of the *Waterlevel* strategy. In the original paper [3] the description was more general. *Waterlevel* is quite involved and consists of a cascade of four algorithms that simulate each other. At the bottom level there is a fractional *Waterlevel* algorithm, denoted by *FWL*, that processes fractional amounts of packets. For any time step t and for any queue q_i, let P_t^i be the set of the last B packets that have arrived at q_i until (and including) time t. Set $P_t = \cup_{i=1}^m P_t^i$. Furthermore, let x_t^p be the extent to which packet p is transmitted by *FWL* during time t. Intuitively, *FWL* tries to serve the available packets as evenly as possible.

Algorithm FWL: At any time t, for any packet $p \in P_t$ match an extent of $x_t^p = \max\{h - s_t^p, 0\}$, where h is the maximum number such that $\sum_{p \in P_t} x_t^p \leq 1$.

We describe an efficient implementation of *FWL*. At any time t and for any packet $p \in P_t$, the algorithm has to determine the extent to which p is served. At any time we maintain a doubly-linked list L of all the s_t^p values, $p \in P_t$, sorted in increasing order. For each entry in the list we store a vector of length m indicating how many packets in q_i, $1 \leq i \leq m$, currently take that value. The values in L together with the total number of packets taking a certain value give rise to a waterlevel profile. Each level of the profile represents a value in L. The width of a level corresponds to the total number of packets $p \in P_t$ having a service extent s_t^p equal to that level. In each time step at which new data packets arrive, we have to update L. This is done by first adding a waterlevel of height $s = 0$ at the head of L, storing for each queue q_i the number n_i of newly arrived packets. If, for queue q_i, the previous load l_i plus n_i exceeds B, then we have to discard the oldest $n_i' = l_i + n_i - B$ packets from q_i. This corresponds to a proper update of P_t. Algorithm *FWL* ensures that packets residing longer in q_i have larger service extents. Thus, starting at the tail of L, we discard for any q_i the oldest n_i' packets. This is done by simply decreasing the number of packets from q_i that contribute to a waterlevel until a total number of n_i' has been discarded. The computation of the x_t^p values amounts to filling water of volume 1 into the waterlevel profile. More specifically, we repeatedly have to find out which adjacent waterlevels to merge. Each merge operation can be performed in $O(m)$ time. Simultaneously, while raising waterlevels, we keep track of the extents X_t^i to which packets from q_i are being served.

The goal of the following steps is to discretize *FWL*. This is done by admitting only full, integral packets to the buffers and by transmitting only full, integral packets. In order to guarantee the same throughput as *FWL* one employs slightly larger buffers. The implementation of the following steps is straightforward.

Algorithm FWL': Work with queues of size $B+1$ and run a simulation of *FWL* on queues of size B. At any time t, in the packet arrival phase, accept as many packets as possible subject to the constraint that only complete packets may be accepted. In the transmission phase, at any time t, transmit a total amount of X_t^i from queue q_i, where X_t^i is the total amount transferred by *FWL* from queue q_i. If $\sum_{i=1}^m X_t^i < 1$, then transmit an amount of $1 - \sum_{i=1}^m X_t^i$ from arbitrary non-empty queues as long as there are such.

Algorithm D(FWL'): Work with queues of size $B+1+\lfloor H_m \rfloor$. Run a simulation of *FWL'* with queues of size $B+1$. At any time t and for any queue q_i, let S_i be the total number of packets transmitted from queue q_i by *D(FWL') before* time t and let S_i' be the total amount of packets from queue q_i transmitted by *FWL'* up to (and including) time t. Transmit a packet from the queue for which the residual service extent $S_i' - S_i$ is largest.

Algorithm Waterlevel: Work with queues of size B. Run a simulation of *D(FWL')*. In each time step, accept a packet if *D(FWL')* accepts it and the corresponding queue is not full. Transmit packets as *D(FWL')* if the corresponding queue is not empty.

Finally, our new algorithm *HSFOD* was already described in Section 2. After each packet arrival phase, for each port, the packet arrival rate has to be updated. This takes linear time.

3.2 Randomized Algorithms

We first present the algorithm *Random Schedule*. In addition to the m packet queues the algorithm maintains m auxiliary queues, each of size B, which are initially empty. Over time the auxiliary queues will contain real numbers from the range $(0, 1)$, which serve as priorities. These priorities may be labeled as either marked or unmarked. In the following, $q_1, \ldots, q_m$ will refer to the original packet queues and $Q_1, \ldots, Q_m$ to the auxiliary queues.

Algorithm Random Schedule: At any time execute the following two steps. (1) For any new packet admitted to a queue q_i, choose a real number uniformly at random from $(0, 1)$ and append it to Q_i. If Q_i was full prior to this operation, then first delete the element at the head of Q_i. The newly inserted number is labeled unmarked. (2) In the transmission phase, check if the $Q_1, \ldots, Q_m$ store unmarked numbers. If so, let Q_i be the queue storing the largest unmarked number; ties may be broken arbitrarily. Change the label of that number to marked and transmit a data packet from queue q_i. Otherwise, if there are no unmarked numbers, transmit a packet from an arbitrary non-empty queue.

We remark that *Random Schedule* uses a considerable amount of $\Theta(mB)$ extra space to store the auxiliary queues $Q_1, \ldots, Q_m$. In our implementation we maintain a priority queue based on standard heaps that stores the unmarked numbers from $Q_1, \ldots, Q_m$. Whenever a new data packet is admitted to a packet buffer q_i, we have to insert a number into the priority queue. This operation may be preceded by a *delete* operation if a number first has to be removed from

the head of Q_i. Executing a *deletemax* operation, we can determine which of the packet buffers to serve. All the operations may take up to $O(\log(mB))$ time.

The second randomized switching algorithm known is called *Random Permutation*. The basic approach of the algorithm is to reduce the packet switching problem with m buffers of size B to one with mB buffers of size 1. To this end, a packet buffer q_i of size B is associated with a set $Q_i = \{q_{i,0}, \ldots, q_{i,B-1}\}$ of B buffers of size 1. A packet arrival sequence σ for the problem with size B buffers is transformed into a sequence $\tilde{\sigma}$ for unit-size buffers by applying a Round Robin policy. More specifically, the j-th packet ever arriving at q_i is mapped to $q_{i,j \bmod B}$ in Q_i. *Random Permutation* at any time runs a simulation of the following algorithm *SimRP* for $m' = mB$ buffers of size 1.

Algorithm SimRP(m'): The algorithm is specified for m' buffers for size 1. Initially, choose a permutation π uniformly at random from the permutations on $\{1, \ldots, m'\}$. In each step transmit the packet from the non-empty queue whose index occurs first in π.

Algorithm Random Permutation: Given a online packet arrival sequence σ for buffer size B, run a simulation of *SimRP(mB)* on $\tilde{\sigma}$. At any time, if *SimRP(mB)* serves a buffer from Q_i, transmit a packet from q_i. If the buffers of *SimRP(mB)* are all empty, transmit a packet from an arbitrary non-empty queue if there is one.

Obviously, the algorithm needs a large amount of $\Theta(mB)$ extra space to run *SimRP(mB)*. Using a priority queue that stores non-empty buffers of capacity 1, we can determine in $O(\log(mB))$ time which queue to serve.

4 The Test Environment

We have tested the online packet buffering algorithms on real-world traces from the Internet Traffic Archive [7], a moderated trace repository maintained by ACM SIGCOMM. We have performed extensive tests with seven traces. A first set of four traces monitors wide-area traffic between Digital Equipment Corporation (DEC) and the rest of the world. A second set of three traces monitors wide-area traffic between the Lawrence Berkeley Laboratory (LBL) and the rest of the world. Only the TCP traffic was considered. The traces were gathered in 1994 and 1995, respectively, over a time horizon of one to two hours and consist of 1.3 to 3.8 million data packets each. Despite being a number of years old, these traces still represent standard benchmarks when marking experimental tests and are recommended for such studies, see e.g. the text book by Krishnamurthy and Rexford [9] or [13]. In the various traces the information relevant to us is, for any data packet, the arrival time and the sending host address.

As indicated in the introduction the main goal of our experiments is to determine the experimentally observed competitiveness of the online switching algorithms and to establish a relative performance ranking among the strategies. Furthermore, we are interested in the algorithms' running times and memory requirements. As for the running time of a strategy, we evaluated the *average time* it takes the al-

gorithm to determine which queue to serve (total running time summed over all time steps/#time steps). As for extra space requirements, we have evaluated, for any of the algorithms, the maximum amount of memory needed by auxiliary data structures employed by that algorithms. Finally in our tests, we have evaluated the actual throughput in terms of the number of data packets transferred.

In our experiments we have studied varying port numbers m as well as varying buffers sizes B. In order to be able to investigate varying values of m, we have to map sending host addresses (e.g. about 3000 in DEC-PKT-1) to port number numbers in the range $\{0, \ldots, m-1\}$. We chose a mapping that maps each sending host address to a port number chosen uniformly at random from $\{0, \ldots, m-1\}$. We would like to point out that such a mapping does not lead to balanced traffic at the ports as some hosts generate a large number of packets. In our traces, under the random mapping, we observe highly non-uniform packet arrival patterns where 10 to 15% of the ports receive ten times as many packets as each of the other ports. This is consistent with the fact that web traffic with respect to packets' source (and destination) addresses is distributed non-uniformly, exhibiting essentially a power-law structure [5,13]. Typically, 10% of the hosts account for 90% of the traffic. However, this fact does not allow a direct conclusion on the distribution of sending hosts among the input ports of a switch. This distribution strongly depends on the network topology. So, alternatively, a power-law governed assignment of hosts to ports is not more reasonable than our uniform distribution.

Another important parameter in the experimental tests is the speed of the switch, i.e. how fast the switch can transfer packets. Here we consider speed values relative to the data volume of a given trace. For a trace data set D, let $f_D =$ (#packets in D)/(length of time horizon of D) be the average packet arrival rate in D. Speed s indicates that the switch forwards data packets with frequency sf_D. Thus, intuitively, a speed 1 switch can forward the data exactly as fast as it arrives on the average. If the speed is low, inevitably, buffers tend to be highly populated. If the speed is high, buffers are only lightly loaded. In summary, each of our experiments is specified by the following parameters: (a) switching algorithm A; (b) trace data set D; (c) number m of buffers; (d) buffer size B; (e) speed s.

5 Experimental Results

We have done extensive tests with all the network traces mentioned in Section 4. A first, very positive finding is that the results are consistent for all the traces. The phenomena reported in this section, unless otherwise stated, have occurred for all the data sets. Due to space limitations, in this paper we only present the results for trace DEC-PKT-1. A zip-file containing the plots for all the traces can be downloaded at `http://www.informatik.uni-freiburg.de/~jacobs/`. In the following subsections we report on the competitiveness, running time and memory requirements of the algorithms as parameters m, B and s vary. It turned out that a variation of the speed s gives the most interesting results and we therefore start with a description of this issue.

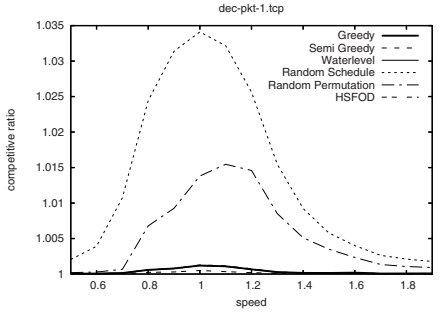


Fig. 1. Comp. ratio, $m = 30$ & $B = 100$

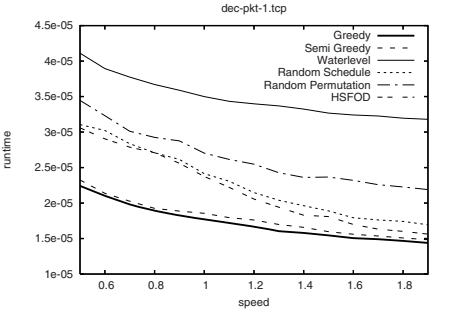


Fig. 2. Run time, $m = 30$ & $B = 100$

5.1 Varying the Speed s

Figure 1 depicts the experimentally observed competitiveness for varying s. We consider in this presentation a basic setting with $m = 30$ and $B = 100$. These parameters are chosen relative to the size of DEC-PKT-1, which consists of 2.1 million packets. More precisely, we wish to simulate the algorithms for sufficiently large m and time steps with considerable packet traffic. Furthermore, switch simulations in the literature usually also work with $m = 8$ to $m = 32$ ports, see e.g. [11,15]. Our basic setting of m and B is not critical. As we will see, the observed phenomena occur for other parameter settings (smaller/larger m and smaller/larger B) as well.

An important result of our study is that the experimentally observed competitiveness of all the algorithms ranges between 1.0 and 1.035 and hence is considerably lower than the theoretical bounds. This is not surprising because competitive analysis is a strong worst-case performance measure. It is astonishing, though, that the gap is so high. Remarkably, *Greedy*, *Semi Greedy* and *Waterlevel* have an experimental competitiveness that is always below 1.002, i.e. they are never 0.2% worse than an optimal solution. *HSFOD* exhibits an even better competitiveness of less than 1.001 for all values of s. Furthermore, interestingly, the curves for *Greedy*, *Semi Greedy* and *Waterlevel* are almost identical and indistinguishable in the plot. The three algorithms have essentially the same performance: For instance, the difference in the number of transferred packets is less than 1000 when the total throughput of each of the three strategies is about 2 million packets. All the algorithms have the highest ratios for values of s around 1. On other traces, the peak sometimes occurs at $s \approx 1.1$. Thus, the worst case occurs when the average packet arrival rate is equal to the rate with which the switch can forward packets and packet scheduling decisions matter. For small and large values of s, the experimental competitiveness tends to 1. This is due to the fact that buffers tend to be either heavily populated (small s) or lightly populated (large s) and all the algorithms transfer essentially an optimum number of packets. Another important result is that the theoretical and experimentally observed competitive ratios are unrelated. In particular, in the

experiments the randomized strategies, which have low theoretical competitive ratios, do perform considerably worse than the deterministic algorithms.

Figure 2 shows the running times of the algorithms, i.e. the average time in seconds it takes an algorithm to perform 1 time step (update auxiliary data structures to account for incoming packets and determine the queue to be served). We evaluate the running times for varying s because the buffer occupancy depends on s and the latter occupancy can affect the running time. Uniformly over all algorithms we observe decreasing running times for increasing values of s. The reason is that, for large s, buffers tend to be empty and the algorithms need less time to handle one time step. *Greedy* and *Semi Greedy* are the fastest algorithms, *Semi-Greedy* being only slightly slower than *Greedy*. *HSFOD*, *Random Schedule*, *Random Permutation* and *Waterlevel* have considerably higher running times. *Waterlevel* is the slowest strategy with running times that are more than twice as high as that of *Greedy*. A shown in Figure 2, the algorithms need 20 to 40 milliseconds to perform one time step. These times would be lower in a switch hardware implementation; our runtime tests just represent a comparative study of the algorithms.

As for the extra memory requirements of the algorithms, the numbers are stable for varying s. The important finding here is that the amounts of extra space differ vastly among the strategies. As to be expected, *HSFOD*, *Greedy* and *Semi Greedy* have small requirements. *HSFOD* uses no more than 500 bytes, while *Greedy* and *Semi Greedy*, using priority queues, allocate 1000 to 1300 bytes. *Waterlevel* has space requirements that are twice as high. Huge amounts of extra space (80.000 to 100.000 bytes) are required by *Random Schedule* and *Random Permutation*. Recall that these algorithms need space for auxiliary queues and mB unit-size buffers.

5.2 Varying the Buffer Size

In a next set of experiments we study the effect of varying the buffer size B. We investigate this effect for the critical speed $s = 1.0$ where the observed competitive ratios are highest. Figure 3 shows that the buffer size has essentially no effect on the competitiveness; only the randomized strategies show a slight fluctuation. This supports our statement that our initial setting $m = 30$ and $B = 100$ plays no particular role. Again, *HSFOD* outperforms all the other algorithms and the performance of *Greedy*, *Semi Greedy* and *Waterlevel* is almost identical. In Figure 4 we observe that the running times, too, are stable. The only exception is *Random Schedule*. The maintenance of its auxiliary queues takes more time as B increases. As for the required space, as was to be expected, the deterministic strategies have fixed demands as the size of the auxiliary data structures depends only on m. *Random Schedule* and *Random Permutation* experience a linear increase as the auxiliary data structures depend on mB. The increase is about 20 bytes per additional buffer cell.

5.3 Varying the Number of Ports

Next we analyze the effect of varying the number m of ports, focusing again on the most critical speed $s = 1.0$. We consider a fixed product of mB, which

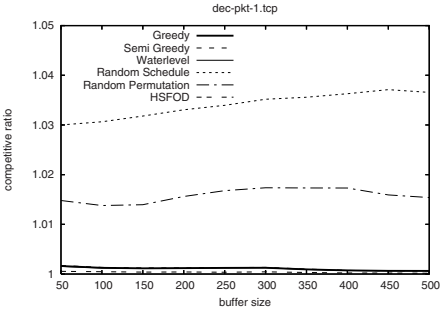


Fig. 3. Comp. ratio, $m = 30$ & $s = 1.0$

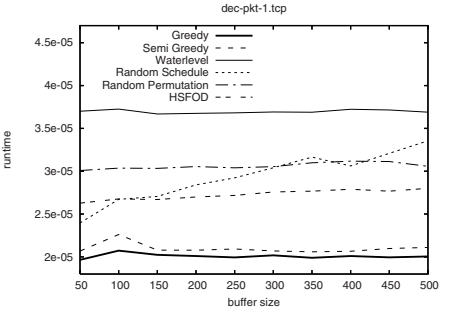


Fig. 4. Run time, $m = 30$ & $s = 1.0$

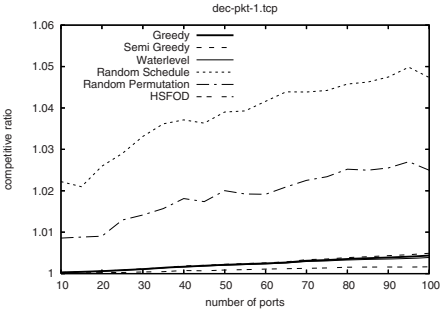


Fig. 5. Comp. ratio, $mB = 3000$ & $s = 1.0$

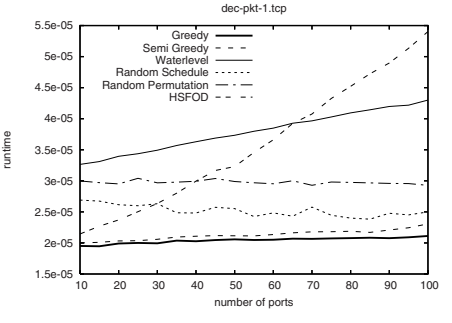


Fig. 6. Run time, $mB = 3000$ & $s = 1.0$

is equal to $mB = 3000$ for our initial parameters. The reason is the following: Varying m while fixing B would investigate the effect to giving a switch more total buffer space, an issue that was already studied in Section 5.2.

Interestingly, the algorithms perform well in our new scenario. All deterministic algorithms show a very slight increase in experimental competitiveness, see Figure 5. The increase is more pronounced in the case of *Random Schedule* and *Random Permutation*. The general increase in competitiveness is due to the fact that for a larger number of ports, online algorithms have a higher chance of serving the "wrong" port. Figure 6 reveals a weakness of *HSFOD*; its running time increases linearly with m. The same holds for *Waterlevel*, although the gradient is smaller here. For all the other strategies the running times are stable. As for the memory requirements, as was to be expected, the deterministic algorithms have slightly increasing demands (about 4 bytes per additional port). The demands are fixed for *Random Schedule* and *Waterlevel* as the sizes of the auxiliary data structures are linear in mB.

5.4 The Absolute Throughput

Finally, we analyze the actual throughput of the algorithms, i.e. the total number of successfully transferred data packets. The corresponding plots appear in the

full paper. We observe an almost linear increase in throughput as s increases, leading to the maximum possible throughput at $s = 1.2$. At our critical speed $s = 1$ we varied again B and m. As B increases, the throughput improves. Increasing the number m of ports while fixing the total amount of memory available in the switch, *SFOD*, *HSFOD*, *Greedy*, *Semi Greedy* and *Waterlevel* experience almost no performance loss. On the other hand, *Random Permutation* and *Random Schedule* experience a loss in throughput.

References

1. Aiello, W., Mansour, Y., Rajagopolan, S., Rosén, A.: Competitive queue policies for differentiated services. In: Proc. INFOCOM, pp. 431–440 (2000)
2. Albers, S., Schmidt, M.: On the performance of greedy algorithms in packet buffering. In: Proc. 36th ACM Symp. on Theory of Computing, pp. 35–44 (2004)
3. Azar, Y., Litichevskey, A.: Maximizing throughput in multi-queue switches. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 53–64. Springer, Heidelberg (2004)
4. Azar, Y., Richter, Y.: Management of multi-queue switches in QoS networks. In: Proc. 35th ACM Symp. on Theory of Computing, pp. 82–89 (2003)
5. Elhanany, I., Chiou, D., Tabatabaee, V., Noro, R., Poursepanj, A.: The network processing forum switch fabric benchmark specifications: An overview. IEEE Network, 5–9 (2005)
6. Embrechts, P., Maejima, M.: Selfsimilar Processes. Princeton Univ. Press, Princeton (2002)
7. The Internet traffic archive, http://ita.ee.lbl.gov
8. Kesselman, A., Lotker, Z., Mansour, Y., Patt-Shamir, B., Schieber, B., Sviridenko, M.: Buffer overflow management in QoS switches. In: Proc. 31st ACM Symp. on Theory of Computing, pp. 520–529 (2001)
9. Krishnamurthy, B., Rexford, J.: Web Protocols and Practice. Addison-Wesley, London, UK (2001)
10. Schmidt, M.: Packet buffering: Randomization beats deterministic algorithms. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, Springer, Heidelberg (2005)
11. Sukhtankar, S., Hecht, D., Rosen, W.: A novel switch architecture for high-performance computing and signal processing networks. In: Proc. 3rd IEEE International Symposium on Network Computing and Applications, pp. 215–222 (2004)
12. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Comm. of the ACM 28, 202–208 (1985)
13. Williamson, C.: Internet traffic measurements. IEEE Internet Computing 5, 70–74 (2001)
14. Willinger, W., Taqqu, M.S., Erramilli, A.: A bibliographical guide to self-similar traffic and performance modeling for modern high-speed networks. In: Kelly, F.P., Zachary, S., Ziedins, I. (eds.) Stochastic Networks Theory and Applications, pp. 339–366. Oxford Science Press, Oxford (1996)
15. Yang, M., Zheng, S.Q.: An efficient scheduling algorithm for CIOQ switches with space-division multiplexing expansion. In: Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM) (2003)

Author Index