Camil Demetrescu (Ed.)

LNCS 4525

Experimental Algorithms

6th International Workshop, WEA 2007
Rome, Italy, June 2007
Proceedings

Lecture Notes in Computer Science 4525

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

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Camil Demetrescu (Ed.)

Experimental Algorithms

6th International Workshop, WEA 2007
Rome, Italy, June 6-8, 2007
Proceedings

Springer

Volume Editor

Camil Demetrescu
Univ. Roma “La Sapienza”
Facoltà di Ingegneria
Via Eudossiana, 18, 00184 Rome, Italy
E-mail: demetres@dis.uniroma1.it

Library of Congress Control Number: 2007927502

CR Subject Classification (1998): F.2.1-2, E.1, G.1-2, I.3.5, I.2.8

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

ISSN 0302-9743
ISBN-10 3-540-72844-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-72844-3 Springer Berlin Heidelberg New York

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

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2007
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12070992 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 6th Workshop on Experimental Algorithms (WEA 2007), held at the School of Engineering of the University of Rome "La Sapienza" on June 6–8, 2007. The conference is devoted to fostering and disseminating high quality research results focused on the experimental analysis of algorithms and aims at bringing together researchers from the computer science and operations research communities. Papers were solicited from all areas of algorithmic engineering research.

The preceding workshops were held in Riga (Latvia, 2001), Ascona (Switzerland, 2003), Angra dos Reis (Brazil, 2004), Santorini (Greece, 2005), and Menorca Island (Spain, 2006). The proceedings of the previous WEAs were published as Springer volumes LNCS 2138 (in conjunction with the 13th International Symposium on Fundamentals of Computation Theory, FCT 2001), LNCS 2647 (2003), LNCS 3059 (2004), LNCS 3503 (2005), and LNCS 4007 (2006).

The conference received 121 submissions. Each submission was reviewed by at least three program committee members and evaluated on its quality, originality, and relevance to the conference. Overall, the program committee wrote 440 reviews with the help of 100 trusted external referees. The committee selected 30 papers, leading to an acceptance rate of 24.8%. On average, the authors of each submitted paper received 800 words of comments. The decision process was made electronically using the EasyChair conference management system.

In addition to the accepted contributions, this volume also contains the invited lectures by Corinna Cortes (Google Research), Peter Sanders (Universität Karlsruhe), and Maria Serna (Universitat Politècnica de Catalunya).

We would like to thank all the authors who responded to the call for papers, the invited speakers, the members of the program committee, as well as the external referees and the organizing committee members.

We gratefully acknowledge support from the University of Rome "La Sapienza" and the University of Rome "Tor Vergata".

April 2007 Camil Demetrescu

Conference Organization

Program Committee

Susanne Albers	U. Freiburg
Eric Angel	U. Evry Val D'Essonne
Giorgio Ausiello	U. Roma “La Sapienza”
Ricardo Baeza-Yates	Yahoo! Research
Jon Bentley	Avaya Labs Research
Adam Buchsbaum	AT&T Labs–Research
Ioannis Chatzigiannakis	U. Patras
Camil Demetrescu (Chair)	U. Roma “La Sapienza”
Cid de Souza	U. Campinas
Rolf Fagerberg	U. Southern Denmark, Odense
Rudolf Fleischer	Fudan U. Shanghai
Monika Henzinger	Google & EPFL
Dorit S. Hochbaum	U. California at Berkeley
Michel Gendreau	U. Montréal
Giuseppe F. Italiano	U. Roma “Tor Vergata”
Riko Jacob	ETH Zürich
Richard Ladner	U. Washington
Kurt Mehlhorn	MPII Saarbrücken
Bernard Moret	EPFL
S. Muthukrishnan	Google
Petra Mutzel	U. Dortmund
Giri Narasimhan	Florida International U.
Robert Sedgewick	Princeton U.
Thomas Stuetzle	U. Libre de Bruxelles
Jan Vahrenhold	U. Dortmund
Renato Werneck	Microsoft Research Silicon Valley
Norbert Zeh	Dalhousie U., Halifax

Steering Committee

Edoardo Amaldi	Politecnico di Milano
David A. Bader	Georgia Tech
Josep Díaz	T.U. Catalonia
Giuseppe F. Italiano	U. Roma “Tor Vergata”
David S. Johnson	AT&T Labs–Research
Klaus Jansen	U. Kiel
Kurt Mehlhorn	MPII Saarbrücken
Ian Munro	U. Waterloo

Sotiris Nikoletseas	U. Patras & CTI
Jose Rolim (Chair)	U. Geneva
Pavlos Spirakis	U. Patras & CTI

Organizing Committee

Vincenzo Bonifaci	U. Roma “La Sapienza”
Saverio Caminiti	U. Roma “La Sapienza”
Fabio Dellutri	U. Roma “Tor Vergata”
Camil Demetrescu	U. Roma “La Sapienza”
Irene Finocchi	U. Roma “La Sapienza”
Luigi Laura	U. Roma “La Sapienza”
Andrea Vitaletti	U. Roma “La Sapienza”

External Reviewers

Ilan Adler
Nina Amenta
Diogo Andrade
Prasanna Balaprakash
Evripidis Bampis
Jørgen Bang-Jensen
Luca Becchetti
Nicola Beume
Immanuel Bomze
Vincenzo Bonifaci
Markus Borschbach
Prosenjit Bose
Patricia Buendia
Luciana Buriol
Pippo Cattaneo
Victor Cavalcante
Bala Chandran
Marco Chiarandini
Markus Chimani
Edmund Christiansen
Alexandre Cunha
Florian Diedrich
David Eppstein
Udo Feldkamp
Carlos Ferreira
Esteban Feuerstein
Michele Flammini
Paolo Giulio Franciosa
Markus Gärtner
Michael Gatto
Fabian Gieseke
Aristides Gionis
Andrew V. Goldberg
Laurent Gourves
Fabrizio Grandoni
Carsten Gutwenger
Edna Hoshino
Frank Hutter
David S. Johnson
Allan Jørgensen
Marcin Kaminski
Maria Kandyba
George Karakostas
Athanasios Kinalis
Karsten Klein
Stephen Kobourov
Sven O. Krumke
Alexander Kröller
Kim S. Larsen
Orlando Lee
Asaf Levin
Vassiliki Liagkou
Andrea Lodi
Manuel López-Ibáñez
Abilio Lucena
Charles (Chip) Martel
Peter Marwedel
Jens Maue

Victor Milenkovic
Michele Monaci
Gabriel Moruz
Arnaldo Moura
Georgios Mylonas
Umberto Nanni
Gonzalo Navarro
Marc Nunkesser
Marco A. Montes de Oca
Rasmus Pagh
Konstantinos Panagiotou
Luis Paquete
Fanny Pascual
Wolfgang Paul
Christian Nørgaard Storm Pedersen
Leon Peeters
Wei Peng
Ulrich Pferschy
Olivier Powell
Kirk Pruhs
Joe Qranfal
Knut Reinert
Andrea Ribichini
Giovanni Rinaldi
Adi Rosen
Domenico Saccà
Claus P. Schnorr
Meera Sitharam
Martin Skutella
Michiel Smid
Damien Stehle
Etsuji Tomita
Jan van der Veen
Andrea Vitaletti
Yoshiko Wakabayashi
Michael Waschbüsch
Anthony Wirth
Hans-Christoph Wirth
Hoi-Ming Wong
Eduardo Xavier
Tallys Yunes
Martin Zachariasen

Table of Contents

Invited Lectures

An Alternative Ranking Problem for Search Engines 1
Corinna Cortes, Mehryar Mohri, and Ashish Rastogi

Engineering Fast Route Planning Algorithms 23
Peter Sanders and Dominik Schultes

Random Models for Geometric Graphs (Abstract) 37
Maria Serna

Session 1 (Route Planning)

Better Landmarks Within Reach 38
Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck

Landmark-Based Routing in Dynamic Graphs 52
Daniel Delling and Dorothea Wagner

Dynamic Highway-Node Routing 66
Dominik Schultes and Peter Sanders

Session 2 (Dynamic Trees, Skip Lists, and Bloom Filters)

Dynamic Trees in Practice 80
Robert E. Tarjan and Renato F. Werneck

On the Cost of Persistence and Authentication in Skip Lists 94
Michael T. Goodrich, Charalampos Papamanthou, and Roberto Tamassia

Cache-, Hash- and Space-Efficient Bloom Filters 108
Felix Putze, Peter Sanders, and Johannes Singler

Session 3 (Crossing Minimizationm, TSP, and Vehicle Routing

Crossing Minimization in Weighted Bipartite Graphs 122
OlcaA. Çakiroğlu, Cesim Erten, Ömer Karataş, and Melih Sözdinler

Fast Minimum-Weight Double-Tree Shortcutting for Metric TSP 136
Vladimir Deineko and Alexander Tiskin

A Robust Branch-Cut-and-Price Algorithm for the Heterogeneous Fleet Vehicle Routing Problem 150
Artur Pessoa, Marcus Poggi de Aragão, and Eduardo Uchoa

Session 4 (Network Routing and Stability)

Simple and Efficient Geographic Routing Around Obstacles for Wireless Sensor Networks 161
Olivier Powell and Sotiris Nikoletseas

A Distributed Primal-Dual Heuristic for Steiner Problems in Networks 175
Marcelo Santos, Lúcia M.A. Drummond, and Eduardo Uchoa

An Experimental Study of Stability in Heterogeneous Networks 189
Maria Chroni, Dimitrios Koukopoulos, and Stavros D. Nikolopoulos

Session 5 (Strings and Range Searching)

Simple Compression Code Supporting Random Access and Fast String Matching 203
Kimmo Fredriksson and Fedor Nikitin

Engineering a Compressed Suffix Tree Implementation 217
Niko Välimäki, Wolfgang Gerlach, Kashyap Dixit, and Veli Mäkinen

Simple Space-Time Trade-Offs for AESA 229
Karina Figueroa and Kimmo Fredriksson

Session 6 (Matching, Flows, and Spanners)

Engineering Algorithms for Approximate Weighted Matching 242
Jens Maue and Peter Sanders

Experimental Evaluation of Parametric Max-Flow Algorithms 256
Maxim Babenko, Jonathan Derryberry, Andrew Goldberg, Robert Tarjan, and Yunhong Zhou

Experimental Study of Geometric t-Spanners: A Running Time Comparison 270
Mohammad Farshi and Joachim Gudmundsson

Session 7 (Covering, Coloring, and Partitioning)

Vertex Cover Approximations on Random Graphs 285
Eyjolfur Asgeirsson and Cli Stein

Optimal Edge Deletions for Signed Graph Balancing 297
Falk Hü ner, Nadja Betzler, and Rolf Niedermeier

Algorithms for the Balanced Edge Partitioning Problem 311
Ekaterina Smorodkina, Mayur Thakur, and Daniel Tauritz

Session 8 (Applications)

Experimental Evaluations of Algorithms for IP Table Minimization..... 324
Angelo Fanelli, Michele Flammini, Domenico Mango, Giovanna Melideo, and Luca Moscardelli

Algorithms for Longer OLED Lifetime 338
Friedrich Eisenbrand, Andreas Karrenbauer, and Chihao Xu

Improving Tree Search in Phylogenetic Reconstruction from Genome Rearrangement Data 352
Fei Ye, Yan Guo, Andrew Lawson, and Jijun Tang

Session 9 (Spanning Trees)

Benchmarks for Strictly Fundamental Cycle Bases 365
Christian Liebchen, Gregor Wünsch, Ekkehard Köhler, Alexander Reich, and Romeo Rizzi

A Primal Branch-and-Cut Algorithm for the Degree-Constrained Minimum Spanning Tree Problem 379
Markus Behle, Michael Jünger, and Frauke Liers

Experimental Analysis of Algorithms for Updating Minimum Spanning Trees on Graphs Subject to Changes on Edge Weights 393
Celso C. Ribeiro and Rodrigo F. Toso

Session 10 (Packing and Auctions)

An Efficient Implementation for the 0-1 Multi-objective Knapsack Problem 406
Cristina Bazgan, Hadrien Hugot, and Daniel Vanderpooten

Trunk Packing Revisited 420
Ernst Althaus, Tobias Baumann, Elmar Schömer, and Kai Werth

Exact Algorithms for the Matrix Bid Auction 433
Dries R. Goossens and Frits C.R. Spieksma

Author Index 447

An Alternative Ranking Problem for Search Engines

Corinna Cortes[1], Mehryar Mohri[2,1], and Ashish Rastogi[2]

[1] Google Research,
76 Ninth Avenue,
New York, NY 10011
[2] Courant Institute of Mathematical Sciences,
251 Mercer Street
New York, NY 10012

Abstract. This paper examines in detail an alternative ranking problem for search engines, movie recommendation, and other similar ranking systems motivated by the requirement to not just accurately predict pairwise ordering but also preserve the magnitude of the preferences or the difference between ratings. We describe and analyze several cost functions for this learning problem and give stability bounds for their generalization error, extending previously known stability results to non-bipartite ranking and magnitude of preference-preserving algorithms. We present algorithms optimizing these cost functions, and, in one instance, detail both a batch and an on-line version. For this algorithm, we also show how the leave-one-out error can be computed and approximated efficiently, which can be used to determine the optimal values of the trade-off parameter in the cost function. We report the results of experiments comparing these algorithms on several datasets and contrast them with those obtained using an AUC-maximization algorithm. We also compare training times and performance results for the on-line and batch versions, demonstrating that our on-line algorithm scales to relatively large datasets with no significant loss in accuracy.

1 Motivation

The learning problem of ranking has gained an increasing amount of interest in the machine learning community over the last decade, in part due to the remarkable success of web search engines and recommender systems (Freund et al., 1998; Crammer & Singer, 2001; Joachims, 2002; Shashua & Levin, 2003; Cortes & Mohri, 2004; Rudin et al., 2005; Agarwal & Niyogi, 2005). The recent Netflix challenge has further stimulated the learning community fueling its research with invaluable datasets (Netflix, 2006).

The goal of information retrieval engines is to return a set of documents, or clusters of documents, ranked in decreasing order of relevance to the user. The order may be common to all users, as with most search engines, or tuned to individuals to provide personalized search results or recommendations. The accuracy of this ordered list is the key quality measure of theses systems.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 1–22, 2007.
© Springer-Verlag Berlin Heidelberg 2007

In most previous research studies, the problem of ranking has been formulated as that of learning from a labeled sample of pairwise preferences a scoring function with small pairwise misranking error (Freund et al., 1998; Herbrich et al., 2000; Crammer & Singer, 2001; Joachims, 2002; Rudin et al., 2005; Agarwal & Niyogi, 2005). But this formulation suffers some short-comings.

Firstly, most users inspect only the top results. Thus, it would be natural to enforce that the results returned near the top be particularly relevant and correctly ordered. The quality and ordering of the results further down the list matter less. An average pairwise misranking error directly penalizes errors at both extremes of a list more heavily than errors towards the middle of the list, since errors at the extremes result in more misranked pairs. However, one may wish to explicitly encode the requirement of ranking quality at the top in the cost function. One common solution is to weigh examples differently during training so that more important or high-quality results be assigned larger weights. This imposes higher accuracy on these examples, but does not ensure a high-quality ordering at the top. A good formulation of this problem leading to a convex optimization problem with a unique minimum is still an open question.

Another shortcoming of the pairwise misranking error is that this formulation of the problem and thus the scoring function learned ignore the magnitude of the preferences. In many applications, it is not sufficient to determine if one example is preferred to another. One may further request an assessment of how large that preference is. Taking this magnitude of preference into consideration is critical, for example in the design of search engines, which originally motivated our study, but also in other recommendation systems. For a recommendation system, one may choose to truncate the ordered list returned where a large gap in predicted preference is found. For a search engine, this may trigger a search in parallel corpora to display more relevant results.

This motivated our study of the problem of ranking while preserving the magnitude of preferences, which we will refer to in short by *magnitude-preserving ranking*.[1] The problem that we are studying bears some resemblance with that of ordinal regression (McCullagh, 1980; McCullagh & Nelder, 1983; Shashua & Levin, 2003; Chu & Keerthi, 2005). It is however distinct from ordinal regression since in ordinal regression the magnitude of the difference in target values is not taken into consideration in the formulation of the problem or the solutions proposed. The algorithm of Chu and Keerthi (2005) does take into account the ordering of the classes by imposing that the thresholds be monotonically increasing, but this still ignores the difference of target values and thus does not follow the same objective. A crucial aspect of the algorithms we propose is that they penalize misranking errors more heavily in the case of larger magnitudes of preferences.

We describe and analyze several cost functions for this learning problem and give stability bounds for their generalization error, extending previously known stability results to non-bipartite ranking and magnitude of preference-preserving algorithms. In particular, our bounds extend the framework of (Bousquet &

[1] This paper is an extended version of (Cortes et al., 2007).

Elisseeff, 2000; Bousquet & Elisseeff, 2002) to the case of cost functions over pairs of examples, and extend the bounds of Agarwal and Niyogi (2005) beyond the bi-partite ranking problem. Our bounds also apply to algorithms optimizing the so-called *hinge rank loss*.

We present several algorithms optimizing these cost functions, and in one instance detail both a batch and an on-line version. For this algorithm, MPRank, we also show how the leave-one-out error can be computed and approximated efficiently, which can be used to determine the optimal values of the trade-off parameter in the cost function. We also report the results of experiments comparing these algorithms on several datasets and contrast them with those obtained using RankBoost (Freund et al., 1998; Rudin et al., 2005), an algorithm designed to minimize the exponentiated loss associated with the Area Under the ROC Curve (AUC), or pairwise misranking. We also compare training times and performance results for the on-line and batch versions of MPRank, demonstrating that our on-line algorithm scales to relatively large datasets with no significant loss in accuracy.

The remainder of the paper is organized as follows. Section 2 describes and analyzes our algorithms in detail. Section 3 presents stability-based generalization bounds for a family of magnitude-preserving algorithms. Section 4 presents the results of our experiments with these algorithms on several datasets.

2 Algorithms

Let S be a sample of m labeled examples drawn i.i.d. from a set X according to some distribution D:

$$(x_1, y_1), \dots, (x_m, y_m) \in X \times \mathbb{R}. \tag{1}$$

For any $i \in [1, m]$, we denote by S^{-i} the sample derived from S by omitting example (x_i, y_i), and by S^i the sample derived from S by replacing example (x_i, y_i) with an other example (x'_i, y'_i) drawn i.i.d. from X according to D. For convenience, we will sometimes denote by $y_x = y_i$ the label of a point $x = x_i \in X$.

The quality of the ranking algorithms we consider is measured with respect to pairs of examples. Thus, a cost functions c takes as arguments two sample points. For a fixed cost function c, the empirical error $\widehat{R}(h, S)$ of a hypothesis $h : X \mapsto \mathbb{R}$ on a sample S is defined by:

$$\widehat{R}(h, S) = \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} c(h, x_i, x_j). \tag{2}$$

The true error $R(h)$ is defined by

$$R(h) = \mathrm{E}_{x,x' \sim D}[c(h, x, x')]. \tag{3}$$

2.1 Cost Functions

We introduce several cost functions related to magnitude-preserving ranking. The first one is the so-called *hinge rank loss* which is a natural extension of the

pairwise misranking loss (Cortes & Mohri, 2004; Rudin et al., 2005). It penalizes a pairwise misranking by the magnitude of preference predicted or the nth power of that magnitude ($n = 1$ or $n = 2$):

$$c^n_{\mathrm{HR}}(h, x, x') = \begin{cases} 0, \text{ if } (h(x') - h(x))(y_{x'} - y_x) \geq 0 \\ \big|(h(x') - h(x))\big|^n, \text{ otherwise.} \end{cases} \tag{4}$$

c^n_{HR} does not take into consideration the true magnitude of preference $y_{x'} - y_x$ for each pair (x, x') however. The following cost function has this property and penalizes deviations of the predicted magnitude with respect to the true one. Thus, it matches our objective of magnitude-preserving ranking ($n = 1, 2$):

$$c^n_{\mathrm{MP}}(h, x, x') = \big|(h(x') - h(x)) - (y_{x'} - y_x)\big|^n. \tag{5}$$

A one-sided version of that cost function penalizing only misranked pairs is given by ($n = 1, 2$):

$$c^n_{\mathrm{HMP}}(h, x, x') = \begin{cases} 0, \text{ if } (h(x') - h(x))(y_{x'} - y_x) \geq 0 \\ \big|(h(x') - h(x)) - (y_{x'} - y_x)\big|^n, \text{ otherwise.} \end{cases} \tag{6}$$

Finally, we will consider the following cost function derived from the ϵ-insensitive cost function used in SVM regression (SVR) (Vapnik, 1998) ($n = 1, 2$):

$$c^n_{\mathrm{SVR}}(h, x, x') = \begin{cases} 0, \text{ if } \big|\big[(h(x') - h(x)) - (y_{x'} - y_x)\big]\big| \leq \epsilon \\ \big|(h(x') - h(x)) - (y_{x'} - y_x) - \epsilon\big|^n, \text{ otherwise.} \end{cases} \tag{7}$$

Note that all of these cost functions are convex functions of $h(x)$ and $h(x')$.

2.2 Objective Functions

The regularization algorithms based on the cost functions c^n_{MP} and c^n_{SVR} correspond closely to the idea of preserving the magnitude of preferences since these cost functions penalize deviations of a predicted difference of score from the target preferences. We will refer by MPRank to the algorithm minimizing the regularization-based objective function based on c^n_{MP}:

$$F(h, S) = \|h\|^2_K + C\frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m} c^n_{\mathrm{MP}}(h, x_i, x_j), \tag{8}$$

and by SVRank to the one based on the cost function c^n_{SVR}

$$F(h, S) = \|h\|^2_K + C\frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m} c^n_{\mathrm{SVR}}(h, x_i, x_j). \tag{9}$$

For a fixed n, $n = 1, 2$, the same stability bounds hold for both algorithms as seen in the following section. However, their time complexity is significantly different.

2.3 MPRank

We will examine the algorithm in the case $n = 2$. Let $\Phi : X \mapsto F$ be the mapping from X to the reproducing Hilbert space. The hypothesis set H that we are considering is that of linear functions h, that is $\forall x \in X, h(x) = w \cdot \Phi(x)$. The objective function can be expressed as follows

$$\begin{aligned} F(h, S) &= \|w\|^2 + C\frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}\big[(w \cdot \Phi(x_j) - w \cdot \Phi(x_i)) - (y_j - y_i)\big]^2 \\ &= \|w\|^2 + \frac{2C}{m}\sum_{i=1}^{m}\|w \cdot \Phi(x_i) - y_i\|^2 - 2C\|w \cdot \bar{\Phi} - \bar{y}\|^2, \end{aligned}$$

where $\bar{\Phi} = \frac{1}{m}\sum_{i=1}^{m}\Phi(x_i)$ and $\bar{y} = \frac{1}{m}\sum_{i=1}^{m} y_i$. The objective function can thus be written with a single sum over the training examples, which results in a more efficient computation of the solution.

Let N be the dimension of the feature space F. For $i = 1, \ldots, m$, let $\mathbf{M}_{x_i} \in \mathbb{R}^{N \times 1}$ denote the column matrix representing $\Phi(x_i)$, $\mathbf{M}_{\bar{\Phi}} \in \mathbb{R}^{N \times 1}$ a column matrix representing $\bar{\Phi}$, $\mathbf{W} \in \mathbb{R}^{N \times 1}$ a column matrix representing the vector w, $\mathbf{M}_Y \in \mathbb{R}^{m \times 1}$ a column matrix whose ith component is y_i, and $\mathbf{M}_{\bar{Y}} \in \mathbb{R}^{m \times 1}$ a column matrix with all its components equal to $\bar{y}$. Let $\mathbf{M}_X, \mathbf{M}_{\bar{X}} \in \mathbb{R}^{N \times m}$ be the matrices defined by:

$$\mathbf{M}_X = [\mathbf{M}_{x_1} \ldots \mathbf{M}_{x_m}] \quad \mathbf{M}_X = [\mathbf{M}_{\bar{\Phi}} \ldots \mathbf{M}_{\bar{\Phi}}]. \tag{10}$$

Then, the expression giving F can be rewritten as

$$F = \|\mathbf{W}\|^2 + \frac{2C}{m}\|\mathbf{M}_X^\top \mathbf{W} - \mathbf{M}_Y\|^2 - \frac{2C}{m}\|\mathbf{M}_{\bar{X}}^\top \mathbf{W} - \mathbf{M}_{\bar{Y}}\|^2.$$

The gradient of F is then given by: $\nabla F = 2\mathbf{W} + \frac{4C}{m}\mathbf{M}_X(\mathbf{M}_X^\top \mathbf{W} - \mathbf{M}_Y) - \frac{4C}{m}\mathbf{M}_{\bar{X}}(\mathbf{M}_{\bar{X}}^\top \mathbf{W} - \mathbf{M}_{\bar{Y}})$. Setting $\nabla F = 0$ yields the unique closed form solution of the convex optimization problem:

$$\mathbf{W} = C'\big(\mathbf{I} + C'(\mathbf{M}_X - \mathbf{M}_{\bar{X}})(\mathbf{M}_X - \mathbf{M}_{\bar{X}})^\top\big)^{-1}(\mathbf{M}_X - \mathbf{M}_{\bar{X}})(\mathbf{M}_Y - \mathbf{M}_{\bar{Y}}), \tag{11}$$

where $C' = \frac{2C}{m}$. Here, we are using the identity $\mathbf{M}_X\mathbf{M}_X^\top - \mathbf{M}_{\bar{X}}\mathbf{M}_{\bar{X}}^\top = (\mathbf{M}_X - \mathbf{M}_{\bar{X}})(\mathbf{M}_X - \mathbf{M}_{\bar{X}})^\top$, which is not hard to verify. This provides the solution of the primal problem. Using the fact the matrices $\big(\mathbf{I} + C'(\mathbf{M}_X - \mathbf{M}_{\bar{X}})(\mathbf{M}_X - \mathbf{M}_{\bar{X}})^\top\big)^{-1}$ and $\mathbf{M}_X - \mathbf{M}_{\bar{X}}$ commute leads to:

$$\mathbf{W} = C'(\mathbf{M}_X - \mathbf{M}_{\bar{X}})\big(\mathbf{I} + C'(\mathbf{M}_X - \mathbf{M}_{\bar{X}})(\mathbf{M}_X - \mathbf{M}_{\bar{X}})^\top\big)^{-1}(\mathbf{M}_Y - \mathbf{M}_{\bar{Y}}). \tag{12}$$

This helps derive the solution of the dual problem. For any $x' \in X$,

$$h(x') = C'\mathbf{K}'(\mathbf{I} + \bar{\mathbf{K}})^{-1}(\mathbf{M}_Y - \mathbf{M}_{\bar{Y}}), \tag{13}$$

where $\mathbf{K}' \in \mathbb{R}^{1 \times m}$ is the row matrix whose jth component is

$$K(x', x_j) - \frac{1}{m}\sum_{k=1}^{m} K(x', x_k) \tag{14}$$

and $\bar{\mathbf{K}}$ is the kernel matrix defined by

$$\frac{1}{C'}(\bar{\mathbf{K}})_{ij} = K(x_i, x_j) - \frac{1}{m}\sum_{k=1}^{m}(K(x_i, x_k) + K(x_j, x_k)) + \frac{1}{m^2}\sum_{k=1}^{m}\sum_{l=1}^{m} K(x_k, x_l),$$

for all $i, j \in [1, m]$. The solution of the optimization problem for MPRank is close to that of a kernel ridge regression problem, but the presence of additional terms makes it distinct, a fact that can also be confirmed experimentally. However, remarkably, it has the same computational complexity, due to the fact that the optimization problem can be written in terms of a single sum, as already pointed out above. The main computational cost of the algorithm is that of the matrix inversion, which can be computed in time $O(N^3)$ in the primal, and $O(m^3)$ in the dual case, or $O(N^{2+\alpha})$ and $O(m^{2+\alpha})$, with $\alpha \approx .376$, using faster matrix inversion methods such as that of Coppersmith and Winograd.

2.4 SVRank

We will examine the algorithm in the case $n = 1$. As with MPRank, the hypothesis set H that we are considering here is that of linear functions h, that is $\forall x \in X, h(x) = w \cdot \Phi(x)$. The constraint optimization problem associated with SVRank can thus be rewritten as

$$\begin{aligned} &\text{minimize} \quad F(h, S) = \|w\|^2 + C\frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}(\xi_{ij} + \xi_{ij}^*) \\ &\text{subject to} \begin{cases} w \cdot (\Phi(x_j) - \Phi(x_i)) - (y_j - y_i) \leq \epsilon + \xi_{ij} \\ (y_j - y_i) - w \cdot (\Phi(x_j) - \Phi(x_i)) \leq \epsilon + \xi_{ij}^* \\ \xi_{ij}, \xi_{ij}^* \geq 0, \end{cases} \end{aligned}$$

for all $i, j \in [1, m]$. Note that the number of constraints are quadratic with respect to the number of examples. Thus, in general, this results in a problem that is more costly to solve than that of MPRank.

Introducing Lagrange multipliers $\alpha_{ij}, \alpha_{ij}^* \geq 0$, corresponding to the first two sets of constraints and $\beta_{ij}, \beta_{ij}^* \geq 0$ for the remaining constraints leads to the following Lagrange function

$$\begin{aligned} L = \|w\|^2 + C\tfrac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}(\xi_{ij} + \xi_{ij}^*) + \\ \sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_{ij}(w \cdot (\Phi(x_j) - \Phi(x_i)) - (y_j - y_i) - \epsilon + \xi_{ij}) + \\ \sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_{ij}^*(-w \cdot (\Phi(x_j) - \Phi(x_i)) + (y_j - y_i) - \epsilon + \xi_{ij}^*) + \\ \sum_{i=1}^{m}\sum_{j=1}^{m}(\beta_{ij}\xi_{ij} + \beta_{ij}^*\xi_{ij}^*). \end{aligned}$$

Taking the gradients, setting them to zero, and applying the Karush-Kuhn-Tucker conditions leads to the following dual maximization problem

$$\begin{aligned}\text{maximize } & \frac{1}{2}\sum_{i,j=1}^{m}\sum_{k,l=1}^{m}(\alpha_{ij}^*-\alpha_{ij})(\alpha_{kl}^*-\alpha_{kl})K_{ij,kl}- \\ & \epsilon\sum_{i,j=1}^{m}(\alpha_{ij}^*-\alpha_{ij})+\sum_{i,j=1}^{m}(\alpha_{ij}^*-\alpha_{ij})(y_j-y_i) \\ \text{subject to } & 0\le\alpha_{ij},\alpha_{ij}^*\le C, \forall i,j\in[1,m],\end{aligned}$$

where $K_{ij,kl} = K(x_i, x_k) + K(x_j, x_l) - K(x_i, x_l) - K(x_j, x_k)$. This quadratic optimization problem can be solved in a way similar to SVM regression (SVR) (Vapnik, 1998) by defining a kernel K' over pairs with $K'((x_i, x_j), (x_k, x_l)) = K_{ij,kl}$, for all $i, j, k, l \in [1, m]$, and associating the target value $y_i - y_j$ to the pair (x_i, x_j).

The computational complexity of the quadratic programming with respect to pairs makes this algorithm less attractive for relatively large samples.

2.5 On-Line Version of MPRank

Recall from Section 2.3 that the cost function for MPRank can be written as

$$F(h,S)=\|w\|^2+\frac{2C}{m}\sum_{i=1}^{m}\Big((w\cdot\Phi(x_i)-y_i)^2-(w\cdot\bar{\Phi}-\bar{y})\Big)^2. \quad (15)$$

This expression suggests that the solution w can be found by solving the following optimization problem

$$\begin{aligned}\underset{w}{\text{minimize}}\ \ & F=\|w\|^2+\frac{2C}{m}\sum_{i=1}^{m}\xi_i^2 \\ \text{subject to } & (w\cdot\Phi(x_i)-y_i)-(w\cdot\bar{\Phi}-\bar{y})=\xi_i \text{ for } i=1,\ldots,m\end{aligned}$$

Introducing the Lagrange multipliers β_i corresponding to the ith equality constraint leads to the following Lagrange function:

$$L(w,\xi,\beta)=\|w\|^2+\frac{2C}{m}\sum_{i=1}^{m}\xi_i^2-\sum_{i=1}^{m}\beta_i\Big((w\cdot\Phi(x_i)-y_i)-(w\cdot\bar{\Phi}-\bar{y})-\xi_i\Big)$$

Setting $\partial L/\partial w = 0$, we obtain $w = \frac{1}{2}\sum_{i=1}^{m}\beta_i(\Phi(x_i)-\bar{\Phi})$, and setting $\partial L/\partial\xi_i = 0$ leads to $\xi_i = -\frac{m}{4C}\beta_i$. Substituting these expression backs in and letting $\alpha_i = \beta_i/2$ result in the optimization problem

$$\underset{\alpha_i}{\text{maximize}}\ -\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j\widetilde{K}(x_i,x_j)-\frac{m}{2C}\sum_{i=1}^{m}\alpha_i^2+2\sum_{i=1}^{m}\alpha_i\widetilde{y_i}, \quad (16)$$

where $\widetilde{K}(x_i,x_j)=K(x_i,x_j)-\frac{1}{m}\sum_{k=1}^{m}(K(x_i,x_k)+K(x_j,x_k))+\frac{1}{m^2}\sum_{k,l=1}^{m}K(x_k,x_l)$

and $\widetilde{y_i} = y_i - \bar{y}$.

Based on the expressions for the partial derivatives of the Lagrange function, we can now describe a gradient descent algorithm that avoids the prohibitive complexity of MPRank that is associated with matrix inversion:

1 **for** $i \leftarrow 1$ **to** m **do** $\alpha_i \leftarrow 0$
2 **repeat**
3 **for** $i \leftarrow 1$ **to** m
4 **do** $\alpha_i \leftarrow \alpha_i + \eta\big[2(\widetilde{y}_i - \sum_{j=1}^m \alpha_j \widetilde{K}(x_i, x_j)) - \frac{m}{C}\alpha_i\big]$
5 **until** convergence

The gradient descent algorithm described above can be straightforwardly modified to an on-line algorithm where points in the training set are processed in T passes, one by one, and the complexity of updates for ith point is $O(i)$ leading to an overall complexity of $O(T \cdot m^2)$. Note that even using the best matrix inversion algorithms, one only achieves a time complexity of $O(m^{2+\alpha})$, with $\alpha \approx .376$. In addition to a favorable complexity if $T = o(m^{.376})$, an appealing aspect of the gradient descent based algorithms is their simplicity. They are quite efficient in practice for datasets with a large number of training points.

2.6 Leave-One-Out Analysis for MPRank

The leave-one-out error of a learning algorithm is typically costly to compute as it in general requires training the algorithm on m subsamples of the original sample. This section shows how the leave-one-out error of MPRank can be computed and approximated efficiently by extending the techniques of Wahba (1990).

The standard definition of the leave-one-out error holds for errors or cost functions defined over a single point. The definition can be extended to cost functions defined over pairs by leaving out each time one pair of points $(x_i, x_j), i \neq j$ instead of a single point.

To simplify the notation, we will denote $h_{S-\{x_i,x_j\}}$ by h_{ij} and $h_{S-\{x,x'\}}$ by $h_{xx'}$. The leave-one-out error of an algorithm L over a sample S returning the hypothesis h_{ij} for a training sample $S - \{x_i, x_j\}$ is then defined by

$$\mathrm{LOO}(L, S) = \frac{1}{m(m-1)} \sum_{i=1}^{m} \sum_{j=1, i \neq j}^{m} c(h_{ij}, x_i, x_j). \quad (17)$$

The following proposition shows that with our new definition, the fundamental property of LOO is preserved.

Proposition 1. *Let $m \geq 2$ and let h' be the hypothesis returned by L when trained over a sample S' of size $m - 2$. Then, the leave-one-out error over a sample S of size m is an unbiased estimate of the true error over a sample of size $m - 2$:*

$$\mathrm{E}_{S \sim D}[\mathrm{LOO}(L, S)] = R(h'), \quad (18)$$

Proof. Since all points of S are drawn i.i.d. and according to the same distribution D,

$$\mathrm{E}_{S\sim D}[\mathrm{LOO}(L,S)] = \frac{1}{m(m-1)} \sum_{i,j=1,i\neq j}^{m} \mathrm{E}_{S\sim D}[c(h_{ij},x_i,x_j)] \tag{19}$$

$$= \frac{1}{m(m-1)} \sum_{x,x'\in S, x\neq x'}^{m} \mathrm{E}_{S\sim D,x,x'\in S}[c(h_{xx'},x,x')] \tag{20}$$

$$= \mathrm{E}_{S\sim D,x,x'\in S}[c(h_{xx'},x,x')] \tag{21}$$

This last term coincides with $\mathrm{E}_{S',x,x'\sim D,|S'|=m-2}[c(h_{xx'},x,x')] = R(h')$. □

In Section 2.3, it was shown that the hypothesis returned by MPRank for a sample S is given by $h(x') = C'\mathbf{K}'(\mathbf{I}+\bar{\mathbf{K}})^{-1}(\mathbf{M}_Y - \mathbf{M}_{\bar{Y}})$ for all $x' \in \mathbf{M}_X$. Let $\mathbf{K}_c$ be the matrix derived from $\mathbf{K}$ by replacing each entry $\mathbf{K}_{ij}$ of $\mathbf{K}$ by the sum of the entries in the same column $\sum_{j=1}^m \mathbf{K}_{ij}$. Similarly, let $\mathbf{K}_r$ be the matrix derived from $\mathbf{K}$ by replacing each entry of $\mathbf{K}$ by the sum of the entries in the same row, and let $\mathbf{K}_{rc}$ be the matrix whose entries all are equal to the sum of all entries of $\mathbf{K}$. Note that the matrix $\bar{\mathbf{K}}$ can be written as:

$$\frac{1}{C'}\bar{\mathbf{K}} = \mathbf{K} - \frac{1}{m}(\mathbf{K}_r + \mathbf{K}_c) + \frac{1}{m^2}\mathbf{K}_{rc}. \tag{22}$$

Let $\mathbf{K}''$ and $\mathbf{U}$ be the matrices defined by:

$$\mathbf{K}'' = \mathbf{K} - \frac{1}{m}\mathbf{K}_r \quad \text{and} \quad \mathbf{U} = C'\mathbf{K}''(\mathbf{I}+\bar{\mathbf{K}})^{-1}. \tag{23}$$

Then, for all $i \in [1,m]$, $h(x_i) = \sum_{k=1}^m \mathbf{U}_{ik}(y_k - \bar{y})$. In the remainder of this section, we will consider the particular case of the $n = 2$ cost function for MPRank, $c^2_{\mathrm{MP}}(h,x_i,x_j) = [(h(x_j)-y_j) - (h(x_i)-y_i)]^2$.

Proposition 2. *Let h' be the hypothesis returned by MPRank when trained on $S-\{x_i,x_j\}$ and let $\bar{h}' = \frac{1}{m}\sum_{k=1}^m h'(x_k)$. For all $i,j \in [1,m]$, let $\mathbf{V}_{ij} = \mathbf{U}_{ij} - \frac{1}{m-2}\sum_{k\notin\{i,j\}} \mathbf{U}_{ik}$. Then, the following identity holds for $c^2_{\mathrm{MP}}(h',x_i,x_j)$.*

$$\begin{aligned}
&\left[(1-\mathbf{V}_{jj})(1-\mathbf{V}_{ii}) - \mathbf{V}_{ij}\mathbf{V}_{ji}\right]^2 c^2_{\mathrm{MP}}(h',x_i,x_j) = \\
&\big[(1-\mathbf{V}_{ii}-\mathbf{V}_{ij})(h(x_j)-y_j) - (1-\mathbf{V}_{ji}-\mathbf{V}_{jj})(h(x_i)-y_i) \\
&-[(1-\mathbf{V}_{ii}-\mathbf{V}_{ij})(\mathbf{V}_{jj}+\mathbf{V}_{ji})(1-\mathbf{V}_{ji}-\mathbf{V}_{jj})(\mathbf{V}_{ii}+\mathbf{V}_{ij})](\bar{h}'-\bar{y})\big]^2 .
\end{aligned} \tag{24}$$

Proof. By Equation 15, the cost function of MPRank can be written as:

$$F = \|w\|^2 + \frac{2C}{m}\sum_{k=1}^m \left[(h(x_k)-y_k) - (\bar{h}-\bar{y})\right]^2, \tag{25}$$

where $\bar{h} = \frac{1}{m}\sum_{k=1}^m h(x_k)$. h' is the solution of the minimization of F when the terms corresponding to x_i and x_j are left out. Equivalently, one can keep these

terms but select new values for y_i and y_j to ensure that these terms are zero. Proceeding this way, the new values y'_i and y'_j must verify the following:

$$h'(x_i) - y'_i = h'(x_j) - y'_j = \bar{h'} - \bar{y'}, \tag{26}$$

with $\bar{y'} = \frac{1}{m}[y'(x_i) + y'(x_j) + \sum_{k \notin \{i,j\}} y_k]$. Thus, by Equation 23, $h'(x_i)$ is given by $h'(x_i) = \sum_{k=1}^{m} \mathbf{U}_{ik}(y_k - \overline{y}')$. Therefore,

$$\begin{aligned}
h'(x_i) - y_i &= \sum_{k \notin \{i,j\}} \mathbf{U}_{ik}(y_k - \overline{y}') + \mathbf{U}_{ii}(y'_i - \overline{y}') + \mathbf{U}_{ij}(y'_j - \overline{y}') - y_i \\
&= \sum_{k \notin \{i,j\}} \mathbf{U}_{ik}(y_k - \overline{y}) - \sum_{k \notin \{i,j\}} \mathbf{U}_{ik}(\overline{y}' - \overline{y}) + \mathbf{U}_{ii}(h'(x_i) - \bar{h'}) \\
&\quad + \mathbf{U}_{ij}(h'(x_j) - \bar{h'}) - y_i \\
&= (h(x_i) - y_i) - \mathbf{U}_{ii}(y_i - \bar{y}) - \mathbf{U}_{ij}(y_j - \bar{y}) - \sum_{k \notin \{i,j\}} \mathbf{U}_{ik}(\overline{y}' - \overline{y}) \\
&\quad + \mathbf{U}_{ii}(h'(x_i) - \bar{h'}) + \mathbf{U}_{ij}(h'(x_j) - \bar{h'}) \\
&= (h(x_i) - y_i) + \mathbf{U}_{ii}(h'(x_i) - y_i) + \mathbf{U}_{ij}(h'(x_j) - y_j) - (\mathbf{U}_{ii} + \mathbf{U}_{ij})(\bar{h'} - \bar{y}) \\
&\quad - \sum_{k \notin \{i,j\}} \mathbf{U}_{ik}(\overline{y}' - \overline{y}) \\
&= (h(x_i) - y_i) + \mathbf{U}_{ii}(h'(x_i) - y_i) + \mathbf{U}_{ij}(h'(x_j) - y_j) - (\mathbf{U}_{ii} + \mathbf{U}_{ij})(\bar{h'} - \bar{y}) \\
&\quad - \sum_{k \notin \{i,j\}} \mathbf{U}_{ik} \frac{1}{m-2}[(h'(x_i) - y_i) + (h'(x_j) - y_j) - 2(\bar{h'} - \bar{y})] \\
&= (h(x_i) - y_i) + \mathbf{V}_{ii}(h'(x_i) - y_i) + \mathbf{V}_{ij}(h'(x_j) - y_j) - (\mathbf{V}_{ii} + \mathbf{V}_{ij})(\bar{h'} - \bar{y}).
\end{aligned}$$

Thus,

$$(1 - \mathbf{V}_{ii})(h'(x_i) - y_i) - \mathbf{V}_{ij}(h'(x_j) - y_j) = (h(x_i) - y_i) - (\mathbf{V}_{ii} + \mathbf{V}_{ij})(\bar{h'} - \bar{y}),$$

Similarly, we have

$$-\mathbf{V}_{ji}(h'(x_i) - y_i) + (1 - \mathbf{V}_{jj})(h'(x_j) - y_j) = (h(x_j) - y_j) - (\mathbf{V}_{jj} + \mathbf{V}_{ji})(\bar{h'} - \bar{y}).$$

Solving the linear system formed by these two equations with unknown variables $(h'(x_i) - y_i)$ and $(h'(x_j) - y_j)$ gives:

$$\begin{aligned}
\big[(1 - \mathbf{V}_{jj})(1 - \mathbf{V}_{ii}) - \mathbf{V}_{ij}\mathbf{V}_{ji}\big](h'(x_i) - y_i) &= (1 - \mathbf{V}_{jj})(h(x_i) - y_i) + \mathbf{V}_{ij}(h(x_j) - y_j) \\
&\quad - [(\mathbf{V}_{ii} + \mathbf{V}_{ij})(1 - \mathbf{V}_{jj}) + (\mathbf{V}_{jj} + \mathbf{V}_{ji})\mathbf{V}_{ij}](\bar{h'} - \bar{y}).
\end{aligned}$$

Similarly, we obtain:

$$\begin{aligned}
\big[(1 - \mathbf{V}_{jj})(1 - \mathbf{V}_{ii}) - \mathbf{V}_{ij}\mathbf{V}_{ji}\big](h'(x_j) - y_j) &= \mathbf{V}_{ji}(h(x_i) - y_i) + (1 - \mathbf{V}_{ii})(h(x_j) - y_j) \\
&\quad - [(\mathbf{V}_{ii} + \mathbf{V}_{ij})\mathbf{V}_{ji} + (\mathbf{V}_{jj} + \mathbf{V}_{ji})(1 - \mathbf{V}_{ii})](\bar{h'} - \bar{y}).
\end{aligned}$$

Taking the difference of these last two equations and squaring both sides yields the expression of $c^2_{\mathrm{MP}}(h', x_i, x_j)$ given in the statement of the proposition. □

Given $\bar{h}'$, Proposition 2 and Equation 17 can be used to compute the leave-one-out error of h efficiently, since the coefficients $\mathbf{U}_{ij}$ can be obtained in time $O(m^2)$ from the matrix $(\mathbf{I}+\bar{\mathbf{K}})^{-1}$ already computed to determine h.

Note that by the results of Section 2.3 and the strict convexity of the objective function, h' is uniquely determined and has a closed form. Thus, unless the points x_i and x_j coincide, the expression $\left[(1-\mathbf{V}_{jj})(1-\mathbf{V}_{ii})-\mathbf{V}_{ij}\mathbf{V}_{ji}\right]$ factor of $c^2_{\mathrm{MP}}(h', x_i, x_j)$ cannot be null. Otherwise, the system of linear equations found in the proof is reduced to a single equation and $h'(x_i)$ (or $h'(x_j)$) is not uniquely specified.

For larger values of m, the average value of h over the sample S should not be much different from that of h', thus we can approximate $\bar{h}'$ by $\bar{h}$. Using this approximation, for a sample with distinct points, we can write for $L =$MPRank

$$\mathrm{LOO}(L,S) \approx \frac{1}{m(m-1)} \sum_{i \neq j} \Bigg[\frac{(1-\mathbf{V}_{ii}-\mathbf{V}_{ij})(h(x_j)-y_j)-(1-\mathbf{V}_{ji}-\mathbf{V}_{jj})(h(x_i)-y_i)}{\left[(1-\mathbf{V}_{jj})(1-\mathbf{V}_{ii})-\mathbf{V}_{ij}\mathbf{V}_{ji}\right]} - \frac{\left[(1-\mathbf{V}_{ii}-\mathbf{V}_{ij})(\mathbf{V}_{jj}+\mathbf{V}_{ji})-(1-\mathbf{V}_{ji}-\mathbf{V}_{jj})(\mathbf{V}_{ii}+\mathbf{V}_{ij})\right]}{\left[(1-\mathbf{V}_{jj})(1-\mathbf{V}_{ii})-\mathbf{V}_{ij}\mathbf{V}_{ji}\right]}(\bar{h}-\bar{y}) \Bigg]^2 .$$

This can be used to determine efficiently the best value of the parameter C based on the leave-one-out error.

Observe that the sum of the entries of each row of $\bar{\mathbf{K}}$ or each row of $\mathbf{K}''$ is zero. Let $\mathbf{M}_1 \in \mathbb{R}^{m\times 1}$ be column matrix with all entries equal to 1. In view of this observation, $\bar{\mathbf{K}}\mathbf{M}_1 = 0$, thus $(\mathbf{I}+\bar{\mathbf{K}})\mathbf{M}_1 = \mathbf{M}_1$, $(\mathbf{I}+\bar{\mathbf{K}})^{-1}\mathbf{M}_1 = \mathbf{M}_1$, and $\mathbf{U}\mathbf{M}_1 = C'\mathbf{K}''(\mathbf{I}+\bar{\mathbf{K}})^{-1}\mathbf{M}_1 = C'\mathbf{K}''\mathbf{M}_1 = 0$. This shows that the sum of the entries of each row of $\mathbf{U}$ is also zero, which yields the following identity for the matrix $\mathbf{V}$:

$$\mathbf{V}_{ij} = \mathbf{U}_{ij} - \frac{1}{m-2} \sum_{k \notin \{i,j\}} \mathbf{U}_{ik} = \mathbf{U}_{ij} + \frac{\mathbf{U}_{ii}+\mathbf{U}_{ij}}{m-2} = \frac{(m-1)\mathbf{U}_{ij}+\mathbf{U}_{ii}}{m-2}. \quad (27)$$

Hence the matrix $\mathbf{V}$ computes

$$\sum_{k=1}^{m} \mathbf{V}_{ik}(y_k - \bar{y}) = \sum_{k=1}^{m} \mathbf{V}_{ik}(y_k-\bar{y}) = \frac{m-1}{m-2} h(x_i). \quad (28)$$

These identities further simplify the expression of matrix $\mathbf{V}$ and its relationship with h.

3 Stability Bounds

Bousquet and Elisseeff (2000) and Bousquet and Elisseeff (2002) gave stability bounds for several regression and classification algorithms. This section shows similar stability bounds for ranking and magnitude-preserving ranking algorithms. This also generalizes the results of Agarwal and Niyogi (2005) which were given in the specific case of bi-partite ranking.

The following definitions are natural extensions to the case of cost functions over pairs of those given by Bousquet and Elisseeff (2002).

Definition 1. *A learning algorithm L is said to be uniformly β-stable with respect to the sample S and cost function c if there exists $\beta \geq 0$ such that for all $S \in (X \times \mathbb{R})^m$ and $i \in [1, m]$,*

$$\forall x, x' \in X, \ |c(h_S, x, x') - c(h_{S^{-i}}, x, x')| \leq \beta. \tag{29}$$

Definition 2. *A cost function c is is σ-admissible with respect to a hypothesis set H if there exists $\sigma \geq 0$ such that for all $h, h' \in H$, and for all $x, x' \in X$,*

$$|c(h, x, x') - c(h', x, x')| \leq \sigma(|\Delta h(x')| + |\Delta h(x)|), \tag{30}$$

with $\Delta h = h' - h$.

3.1 Magnitude-Preserving Regularization Algorithms

For a cost function c such as those just defined and a regularization function N, a regularization-based algorithm can be defined as one minimizing the following objective function:

$$F(h, S) = N(h) + C \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} c(h, x_i, x_j), \tag{31}$$

where $C \geq 0$ is a constant determining the trade-off between the emphasis on the regularization term versus the error term. In much of what follows, we will consider the case where the hypothesis set H is a reproducing Hilbert space and where N is the squared norm in a that space, $N(h) = \|h\|_K^2$ for a kernel K, though some of our results can straightforwardly be generalized to the case of an arbitrary convex N. By the reproducing property, for any $h \in H$, $\forall x \in X,\ h(x) = \langle h, K(x, .)\rangle$ and by Cauchy-Schwarz's inequality,

$$\forall x \in X, |h(x)| \leq \|h\|_K \sqrt{K(x, x)}. \tag{32}$$

Assuming that for all $x \in X, K(x, x) \leq \kappa^2$ for some constant $\kappa \geq 0$, the inequality becomes: $\forall x \in X, |h(x)| \leq \kappa \|h\|_K$. With the cost functions previously discussed, the objective function F is then strictly convex and the optimization problem admits a unique solution. In what follows, we will refer to the algorithms minimizing the objective function F with a cost function defined in the previous section as *magnitude-preserving regularization algorithms*.

Lemma 1. *Assume that the hypotheses in H are bounded, that is for all $h \in H$ and $x \in S$, $|h(x) - y_x| \leq M$. Then, the cost functions c^n_{HR}, c^n_{MP}, c^n_{HMP}, and c^n_{SVR} are all σ_n-admissible with $\sigma_1 = 1$, $\sigma_2 = 4M$.*

Proof. We will give the proof in the case of c^n_{MP}, $n = 1, 2$, the other cases can be treated similarly.

By definition of c^1_{MP}, for all $x, x' \in X$,

$$|c^1_{\mathrm{MP}}(h', x, x') - c^1_{\mathrm{MP}}(h, x, x')| = \big||(h'(x') - h'(x)) - (y_{x'} - y_x)| - |(h(x') - h(x)) - (y_{x'} - y_x)|\big|. \tag{33}$$

Using the identity $\big||X' - Y| - |X - Y|\big| \leq |X' - X|$, valid for all $X, X', Y \in \mathbb{R}$, it follows that

$$|c^1_{\text{MP}}(h', x, x') - c^1_{\text{MP}}(h, x, x')| \leq |\Delta h(x') - \Delta h(x)| \tag{34}$$
$$\leq |\Delta h(x')| + |\Delta h(x)|, \tag{35}$$

which shows the σ-admissibility of c^1_{MP} with $\sigma = 1$. For c^2_{MP}, for all $x, x' \in X$,

$$|c^2_{\text{MP}}(h', x, x') - c^2_{\text{MP}}(h, x, x')| = ||(h'(x') - h'(x)) - (y_{x'} - y_x)|^2 \tag{36}$$
$$-|(h(x') - h(x)) - (y_{x'} - y_x)|^2|$$
$$\leq |\Delta h(x') - \Delta h(x)|(|h'(x') - y_{x'}| + \tag{37}$$
$$|h(x') - y_{x'}| + |h'(x) - y_x| + |h(x) - y_x|)$$
$$\leq 4M(|\Delta h(x')| + |\Delta h(x)|), \tag{38}$$

which shows the σ-admissibility of c^2_{MP} with $\sigma = 4M$. □

Proposition 3. *Assume that the hypotheses in H are bounded, that is for all $h \in H$ and $x \in S$, $|h(x) - y_x| \leq M$. Then, a magnitude-preserving regularization algorithm as defined above is β-stable with $\beta = \frac{4C\sigma_n^2\kappa^2}{m}$.*

Proof. Fix the cost function to be c, one of the σ_n-admissible cost function previously discussed. Let h_S denote the function minimizing $F(h, S)$ and $h_{S^{-k}}$ the one minimizing $F(h, S^{-k})$. We denote by $\Delta h_S = h_{S^{-k}} - h_S$.

Since the cost function c is convex with respect to $h(x)$ and $h(x')$, $\widehat{R}(h, S)$ is also convex with respect to h and for $t \in [0, 1]$,

$$\widehat{R}(h_S + t\Delta h_S, S^{-k}) - \widehat{R}(h_S, S^{-k}) \leq t\left[\widehat{R}(h_{S^{-k}}, S^{-k}) - \widehat{R}(h_S, S^{-k})\right]. \tag{39}$$

Similarly,

$$\widehat{R}(h_{S^{-k}} - t\Delta h_S, S^{-k}) - \widehat{R}(h_{S^{-k}}, S^{-k}) \leq t\left[\widehat{R}(h_S, S^{-k}) - \widehat{R}(h_{S^{-k}}, S^{-k})\right]. \tag{40}$$

Summing these inequalities yields

$$\widehat{R}(h_S + t\Delta h_S, S^{-k}) - \widehat{R}(h_S, S^{-k}) + \widehat{R}(h_{S^{-k}} - t\Delta h_S, S^{-k}) - \widehat{R}(h_{S^{-k}}, S^{-k}) \leq 0. \tag{41}$$

By definition of h_S and $h_{S^{-k}}$ as functions minimizing the objective functions, for all $t \in [0, 1]$,

$$F(h_S, S) - F(h_S + t\Delta h_S, S) \leq 0 \text{ and } F(h_{S^{-k}}, S^{-k}) - F(h_{S^{-k}} - t\Delta h_S, S^{-k}) \leq 0. \tag{42}$$

Multiplying Inequality 41 by C and summing it with the two Inequalities 42 lead to

$$A + \|h_S\|_K^2 - \|h_S + t\Delta h_S\|_K^2 + \|h_{S^{-k}}\|_K^2 - \|h_{S^{-k}} - t\Delta h_S\|_K^2 \leq 0. \tag{43}$$

with $A = C\Big(\widehat{R}(h_S, S) - \widehat{R}(h_S, S^{-k}) + \widehat{R}(h_S + t\Delta h_S, S^{-k}) - \widehat{R}(h_S + t\Delta h_S, S)\Big)$. Since

$$A = \frac{C}{m^2}\Big[\sum_{i \neq k} c(h_S, x_i, x_k) - c(h_S + t\Delta h_S, x_i, x_k) + \sum_{i \neq k} c(h_S, x_k, x_i) - c(h_S + t\Delta h_S, x_k, x_i)\Big], \tag{44}$$

by the σ_n-admissibility of c,

$$|A| \leq \frac{2Ct\sigma_n}{m^2} \sum_{i \neq k} (|\Delta h_S(x_k)| + |\Delta h_S(x_i)|) \leq \frac{4Ct\sigma_n\kappa}{m} \|\Delta h_S\|_K.$$

Using the fact that $\|h\|_K^2 = \langle h, h\rangle$ for any h, it is not hard to show that

$$\|h_S\|_K^2 - \|h_S + t\Delta h_S\|_K^2 + \|h_{S^{-k}}\|_K^2 - \|h_{S^{-k}} - t\Delta h_S\|_K^2 = 2t(1-t)\|\Delta h_S\|_K^2.$$

In view of this and the inequality for $|A|$, Inequality 43 implies $2t(1-t)\|\Delta h_S\|_K^2 \leq \frac{4Ct\sigma_n\kappa}{m}\|\Delta h_S\|_K$, that is after dividing by t and taking $t \to 0$,

$$\|\Delta h_S\|_K \leq \frac{2C\sigma_n\kappa}{m}. \tag{45}$$

By the σ_n-admissibility of c, for all $x, x' \in X$,

$$\begin{aligned} |c(h_S, x, x') - c(h_{S^{-k}}, x, x')| &\leq \sigma_n(|\Delta h_S(x')| + |\Delta h_S(x)|) && (46)\\ &\leq 2\sigma_n\kappa\|\Delta h_S\|_K && (47)\\ &\leq \frac{4C\sigma_n^2\kappa^2}{m}. && (48) \end{aligned}$$

This shows the β-stability of the algorithm with $\beta = \frac{4C\sigma_n^2\kappa^2}{m}$. □

To shorten the notation, in the absence of ambiguity, we will write in the following $\widehat{R}(h_S)$ instead of $\widehat{R}(h_S, S)$.

Theorem 1. *Let c be any of the cost functions defined in Section 2.1. Let L be a uniformly β-stable algorithm with respect to the sample S and cost function c and let h_S be the hypothesis returned by L. Assume that the hypotheses in H are bounded, that is for all $h \in H$, sample S, and $x \in S$, $|h(x) - y_x| \leq M$. Then, for any $\epsilon > 0$,*

$$\Pr_{S \sim D}\Big[|R(h_S) - \widehat{R}(h_S)| > \epsilon + 2\beta\Big] \leq 2e^{-\frac{m\epsilon^2}{2(\beta m + (2M)^n)^2}}. \tag{49}$$

Proof. We apply McDiarmid's inequality (McDiarmid, 1998) to $\Phi(S) = R(h_S) - \widehat{R}(h_S, S)$. We will first give a bound on $\mathrm{E}[\Phi(S)]$ and then show that $\Phi(S)$ satisfies the conditions of McDiarmid's inequality.

We will denote by $S^{i,j}$ the sample derived from S by replacing x_i with x'_i and x_j with x'_j, with x'_i and x'_j sampled i.i.d. according to D.

Since the sample points in S are drawn in an i.i.d. fashion, for all $i, j \in [1, m]$,

$$\mathrm{E}_S[\widehat{R}(h_S, S)] = \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}\mathrm{E}[c(h_S, x_i, x_j)] \tag{50}$$
$$= \mathrm{E}_{S\sim D}[c(h_S, x_i, x_j)] \tag{51}$$
$$= \mathrm{E}_{S^{i,j}\sim D}[c(h_{S^{i,j}}, x'_i, x'_j)] \tag{52}$$
$$= \mathrm{E}_{S, x'_i, x'_j \sim D}[c(h_{S^{i,j}}, x'_i, x'_j)]. \tag{53}$$

Note that by definition of $R(h_S)$, $\mathrm{E}_S[R(h_S)] = \mathrm{E}_{S,x'_i,x'_j\sim D}[c(h_S, x'_i, x'_j)]$. Thus, $\mathrm{E}_S[\Phi(S)] = \mathrm{E}_{S,x,x'}[c(h_S, x'_i, x'_j) - c(h_{S^{i,j}}, x'_i, x'_j)]$, and by β-stability (Proposition 3)

$$|\,\mathrm{E}_S[\Phi(S)]| \leq \mathrm{E}_{S,x,x'}[|c(h_S, x'_i, x'_j) - c(h_{S^i}, x'_i, x'_j)|] + \tag{54}$$
$$\mathrm{E}_{S,x,x'}[|c(h_{S^i}, x'_i, x'_j) - c(h_{S^{i,j}}, x'_i, x'_j)|] \tag{55}$$
$$\leq 2\beta. \tag{56}$$

Now,

$$|R(h_S) - R(h_{S^k})| = |\,\mathrm{E}_S[c(h_S, x, x') - c(h_{S^k}, x, x')]| \tag{57}$$
$$\leq \mathrm{E}_S[|c(h_S, x, x') - c(h_{S^k}, x, x')|] \tag{58}$$
$$\leq \beta. \tag{59}$$

For any $x, x' \in X$, $|c(h_S, x_k, x_j) - c(h_{S^k}, x_i, x'_k)| < \mathrm{E}_S[|c(h_{S^k}, x, x') - c(h_{S^k}, x, x')|] \leq (2M)^n$, where $n = 1$ or $n = 2$. Thus, we have

$$|\widehat{R}(h_S) - \widehat{R}(h_S^k)| \leq \frac{1}{m^2}\sum_{i\neq k}\sum_{j\neq k}|c(h_S, x_i, x_j) - c(h_{S^k}, x_i, x_j)| + \tag{60}$$
$$\frac{1}{m^2}\sum_{j=1}^{m}|c(h_S, x_k, x_j) - c(h_{S^k}, x'_k, x_j)| + \tag{61}$$
$$\frac{1}{m^2}\sum_{i=1}^{m}|c(h_S, x_k, x_j) - c(h_{S^k}, x_i, x'_k)| \tag{62}$$
$$\leq \frac{1}{m^2}(m^2\beta) + \frac{m}{m^2}2(2M)^n = \beta + 2(2M)^n/m. \tag{63}$$

Thus,

$$|\Phi(S) - \Phi(S^k)| \leq 2(\beta + (2M)^n/m), \tag{64}$$

and $\Phi(S)$ satisfies the hypotheses of McDiarmid's inequality. □

The following Corollary gives stability bounds for the generalization error of magnitude-preserving regularization algorithms.

Corollary 1. *Let L be a magnitude-preserving regularization algorithm and let c be the corresponding cost function and assume that for all $x \in X, K(x, x) \leq \kappa^2$. Assume that the hypothesis set H is bounded, that is for all $h \in H$, sample S, and $x \in S$, $|h(x) - y_x| \leq M$. Then, with probability at least $1 - \delta$,*

– *for* $n = 1$,

$$R(h_S) \leq \widehat{R}(h_S) + \frac{8\kappa^2 C}{m} + 2(2\kappa^2 C + M)\sqrt{\frac{2}{m}\log\frac{2}{\delta}}; \qquad (65)$$

– *for* $n = 2$,

$$R(h_S) \leq \widehat{R}(h_S) + \frac{128\kappa^2 C M^2}{m} + 4M^2(16\kappa^2 C + 1)\sqrt{\frac{2}{m}\log\frac{2}{\delta}}. \qquad (66)$$

Proof. By Proposition 3, these algorithms are β-stable with $\beta = \frac{4C\sigma_n^2\kappa^2}{m}$. □

These bounds are of the form $R(h_S) \leq \widehat{R}(h_S) + O(\frac{C}{\sqrt{m}})$. Thus, they are effective for values of $C \ll \sqrt{m}$.

4 Experiments

In this section, we report the results of experiments with two of our magnitude-preserving algorithms, MPRank and SVRank.

The algorithms were tested on four publicly available data sets, three of which are commonly used for collaborative filtering: MovieLens, Book-Crossings, and Jester Joke. The fourth data set is the Netflix data. The first three datasets are available from the following URL:

`http://www.grouplens.org/taxonomy/term/14`.

The Netflix data set is available at

`http://www.netflixprize.com/download`.

4.1 MovieLens Dataset

The MovieLens dataset consists of approximately 1M ratings by 6,040 users for 3,900 movies. Ratings are integers in the range of 1 to 5. For each user, a different predictive model is designed. The ratings of that user on the 3,900 movies (not all movies will be rated) form the target values y_i. The other users' ratings of the ith movie form the ith input vector x_i.

We followed the experimental set-up of Freund et al. (1998) and grouped the reviewers according to the number of movies they have reviewed. The groupings were $20-40$ movies, $40-60$ movies, and $60-80$ movies.

Test reviewers were selected among users who had reviewed between 50 and 300 movies. For a given test reviewer, 300 reference reviewers were chosen at random from one of the three groups and their rating were used to form the input vectors. Training was carried out on half of the test reviewer's movie ratings and testing was performed on the other half. The experiment was done for 300 different test reviewers and the average performance recorded. The whole

Table 1. Performance results for MPRank, SVRank, and RankBoost

Data set	Mean Squared Difference			Mean 1-Norm Difference		
	MPRank	SVRank	RBoost	MPRank	SVRank	RBoost
MovieLens 20-40	2.01 ± 0.02	2.43 ± 0.13	12.88 ± 2.15	1.04 ± 0.05	1.17 ± 0.03	2.59 ± 0.04
MovieLens 40-60	2.02 ± 0.06	2.36 ± 0.16	20.06 ± 2.76	1.04 ± 0.02	1.15 ± 0.07	2.99 ± 0.12
MovieLens 60-80	2.07 ± 0.05	2.66 ± 0.09	21.35 ± 2.71	1.06 ± 0.01	1.24 ± 0.02	3.82 ± 0.23
Jester 20-40	51.34 ± 2.90	55.00 ± 5.14	77.08 ± 17.1	5.08 ± 0.15	5.40 ± 0.20	5.97 ± 0.16
Jester 40-60	46.77 ± 2.03	57.75 ± 5.14	80.00 ± 18.2	4.98 ± 0.13	5.27 ± 0.20	6.18 ± 0.11
Jester 60-80	49.33 ± 3.11	56.06 ± 4.26	88.61 ± 18.6	4.88 ± 0.14	5.25 ± 0.19	6.46 ± 0.20
Netflix Density:32%	1.58 ± 0.04	1.80 ± 0.05	57.5 ± 7.8	0.92 ± 0.01	0.95 ± 0.02	6.48 ± 0.55
Netflix Density:46%	1.55 ± 0.03	1.90 ± 0.06	23.9 ± 2.9	0.95 ± 0.01	1.02 ± 0.02	4.10 ± 0.23
Netflix Density:58%	1.49 ± 0.03	1.93 ± 0.06	12.33 ± 1.47	0.94 ± 0.01	1.06 ± 0.02	3.01 ± 0.15
Books	4.00 ± 3.12	3.64 ± 3.04	7.58 ± 9.95	1.38 ± 0.60	1.32 ± 0.56	1.72 ± 1.05

process was then repeated ten times with a different set of 300 reviewers selected at random. We report mean values and standard deviation for these ten repeated experiments for each of the three groups. Missing review values in the input features were populated with the median review score of the given reference reviewer.

4.2 Jester Joke Dataset

The Jester Joke Recommender System dataset contains 4.1M continuous ratings in the range -10.00 to +10.00 of 100 jokes from 73,496 users. The experiments were set up in the same way as for the MovieLens dataset.

4.3 Netflix Dataset

The Netflix dataset contains more than 100M ratings by 480,000 users for 17,700 movies. Ratings are integers in the range of 1 to 5. We constructed three subsets of the data with different user densities. Subsets were obtained by thresholding against two parameters: the minimum number of movies rated by a user and

the minimum of ratings for a movie. Thus, in choosing users for the training and testing set, we only consider those users who have reviewed more than 150, 500, or 1500 movies respectively. Analogously, in selecting the movies that would appear in the subset data, we only consider those movies that have received at least $360, 1200$, or 1800 reviews. The experiments were then set-up in the same way as for the MovieLens dataset. The mean densities of the three subsets (across the ten repetitions) were $32\%, 46\%$ and 58% respectively. Finally, the test raters were selected from a mixture of the three densities.

4.4 Book-Crossing Dataset

The book-crossing dataset contains 1,149,780 ratings for 271,379 books for a group of 278,858 users. The low density of ratings makes predictions very noisy in this task. Thus, we required users to have reviewed at least 200 books, and then only kept books with at least 10 reviews. This left us with a dataset of 89 books and 131 reviewers. For this dataset, each of the 131 reviewers was in turn selected as a test reviewer, and the other 130 reviewers served as input features. The results reported are mean values and standard deviations over these 131 leave-one-out experiments.

4.5 Performance Measures and Results

The performance measures we report correspond to the problem we are solving. The cost function of MPRank is designed to minimize the squared difference between all pairs of target values, hence we report the mean squared difference (MSD) over all pairs in the test set of size m' of a hypothesis h:

$$\frac{1}{m'^2} \sum_{i=1}^{m'} \sum_{j=1}^{m'} \left((h(x_j) - h(x_i)) - (y_j - y_i)\right)^2 . \tag{67}$$

The cost function of SVRank minimizes the absolute value of the difference between all pairs of examples, hence we report the average of the 1-norm difference, M1D:

$$\frac{1}{m'^2} \sum_{i=1}^{m'} \sum_{j=1}^{m'} \left|(h(x_j) - h(x_i)) - (y_j - y_i)\right| . \tag{68}$$

The results for MPRank and SVRank are obtained using Gaussian kernels. The width of the kernel and the other cost function parameters were first optimized on a held-out sample. The performance on their respective cost functions was optimized and the parameters fixed at these values.

The results are reported in Table 1. They demonstrate that the magnitude-preserving algorithms are both successful at minimizing their respective objective. MPRank obtains the best MSD values and the two algorithms obtain comparable M1D values. However, overall, in view of these results and the superior computational efficiency of MPRank already pointed out in the previous section, we consider MPRank as the best performing algorithm for such tasks.

Table 2. Comparison of MPRank and RankBoost for pairwise misrankings

Data set	Pairwise Misrankings	
	MPRank	RBoost
MovieLens 40-60	0.471 ± 0.005	0.476 0 ± 0.007
MovieLens 60-80	0.442 ± 0.005	0.463 ± 0.011
Jester 20-40	0.414 ± 0.005	0.479 ± 0.008
Jester 40-60	0.418 ± 0.007	0.432 ± 0.005
Netflix Density:32%	0.433 ± 0.018	0.447 ± 0.027
Netflix Density:46%	0.368 ± 0.014	0.327 ± 0.008
Netflix Density:58%	0.295 ± 0.006	0.318 ± 0.008

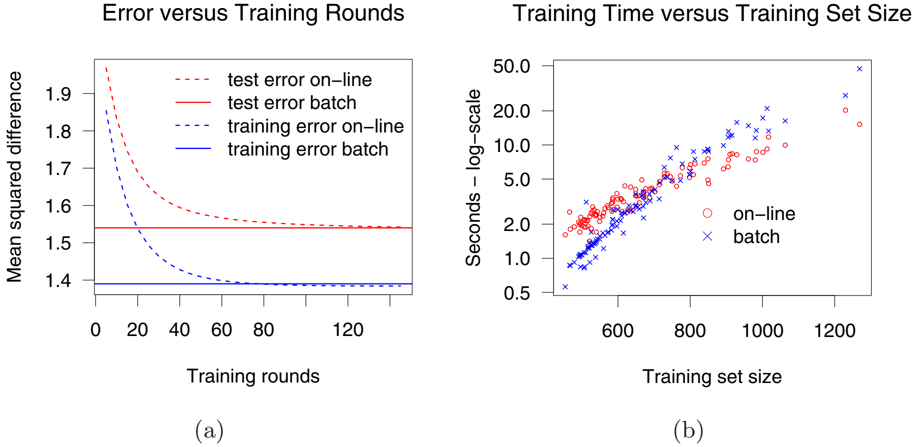
To further examine the ranking properties of MPRank we conducted a number of experiments where we compared the pairwise misranking performance of the algorithm to that of RankBoost, an algorithm designed to minimize the number of pairwise misrankings (Rudin et al., 2005). We used the same features for RankBoost as for MPRank that is we used as weak rankers threshold functions over other reviewers' ratings. As for the other algorithms, the parameter of RankBoost, that is the number of boosting rounds required to minimize pairwise misranking was determined on a held-out sample and then fixed at this value.

Table 2 shows a comparison between these two algorithms. It reports the fraction of pairwise misrankings for both algorithms using the same experimental set-up as previously described:

$$\frac{\sum_{i,j=1}^{m'} 1_{y_i > y_j \wedge h(x_i) \leq h(x_j)}}{\sum_{i,j=1}^{m'} 1_{y_i > y_j}}. \tag{69}$$

The results show that the pairwise misranking error of MPRank is comparable to that of RankBoost. This further increases the benefits of MPRank as a ranking algorithm.

We also tested the performance of RankBoost with respect to MSD and M1D (see Table 1). Naturally, RankBoost is not designed to optimize these performance measure and does not lead to competitive results with respect to MPRank and SVRank on any of the datasets examined.



(a) (b)

Fig. 1. (a) Convergence of the on-line learning algorithm towards the batch solution. Rounding errors give rise to slightly different solutions. (b) Training time in seconds for the on-line and the batch algorithm. For small training set sizes the batch version is fastest, but for larger training set sizes the on-line version is faster. Eventually the batch version becomes infeasible.

4.6 On-Line Version of MPRank

Using the Netflix data we also experimented with the on-line version of MPRank described in Section 2.5. The main questions we wished to investigate were the convergence rate and CPU time savings of the on-line version with respect to the batch algorithm MPRank (Equation 13). The batch solution requires a matrix inversion and becomes infeasible for large training sets.

Figure 1(a) illustrates the convergence rate for a typical reviewer. In this instance, the training and test sets each consisted of about 700 movies. As can be seen from the plot, the on-line version converges to the batch solution in about 120 rounds, where one round is a full cycle through the training set.

Based on monitoring several convergence plots, we decided on terminating learning in the on-line version of MPRank when consecutive rounds of iterations over the full training set would change the cost function by less than .01 %. Figure 1(b) compares the CPU time for the on-line version of MPRank with the batch solution. For both computations of the CPU times, the time to construct the Gram matrix is excluded. The figure shows that the on-line version is significantly faster for large datasets, which extends the applicability of our algorithms beyond the limits of intractable matrix inversion.

5 Conclusion

We presented several algorithms for magnitude-preserving ranking problems and provided stability bounds for their generalization error. We also reported the

results of several experiments on public datasets comparing these algorithms. We presented an on-line version of one of the algorithms and demonstrated its applicability for very large data sets. We view accurate magnitude-preserving ranking as an important problem for improving the quality of modern recommendation and rating systems. An alternative for incorporating the magnitude of preferences in cost functions is to use weighted AUC, where the weights reflect the magnitude of preferences and extend existing algorithms. This however, does not exactly coincide with the objective of preserving the magnitude of preferences.

Acknowledgments

The work of Mehryar Mohri and Ashish Rastogi was partially funded by the New York State Office of Science Technology and Academic Research (NYSTAR). This project was also sponsored in part by the Department of the Army Award Number W81XWH-04-1-0307. The U.S. Army Medical Research Acquisition Activity, 820 Chandler Street, Fort Detrick MD 21702-5014 is the awarding and administering acquisition office. The content of this material does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Bibliography

Agarwal, S., Niyogi, P.: Stability and generalization of bipartite ranking algorithms. In: Auer, P., Meir, R. (eds.) COLT 2005. LNCS (LNAI), vol. 3559, Springer, Heidelberg (2005)

Bousquet, O., Elisseeff, A.: Algorithmic stability and generalization performance. Advances in Neural Information Processing Systems (NIPS 2000) (2000)

Bousquet, O., Elisseeff, A.: Stability and generalization. J. Mach. Learn. Res. 2, 499–526 (2002)

Chu, W., Keerthi, S.S.: New approaches to support vector ordinal regression. In: Proceedings of the 22nd International Conference on Machine Learning, pp. 145–152. ACM Press, New York, USA (2005)

Cortes, C., Mohri, M.: AUC Optimization vs. Error Rate Minimization. In: Advances in Neural Information Processing Systems (NIPS 2003), MIT Press, Vancouver, Canada (2004)

Cortes, C., Mohri, M., Rastogi, A.: Magnitude-preserving ranking algorithms (Technical Report TR-2007-887). Courant Institute of Mathematical Sciences, New York University (2007)

Crammer, K., Singer, Y.: Pranking with ranking. Advances in Neural Information Processing Systems (NIPS 2001) (2001)

Freund, Y., Iyer, R., Schapire, R.E., Singer, Y.: An efficient boosting algorithm for combining preferences. In: Proceedings of the 15th International Conference on Machine Learning, pp. 170–178. Morgan Kaufmann, Madison, San Francisco, US (1998)

Herbrich, R., Graepel, T., Obermayer, K.: Large margin rank boundaries for ordinal regression. In: Smola, Bartlett, Schoelkopf, Schuurmans (eds.) Advances in large margin classifiers, MIT Press, Cambridge, MA (2000)

Joachims, T.: Evaluating retrieval performance using clickthrough data (2002)

McCullagh, P.: Regression models for ordinal data. Journal of the Royal Statistical Society B, 42 (1980)

McCullagh, P., Nelder, J.A.: Generalized linear models. Chapman & Hall, London (1983)

McDiarmid, C.: Concentration. Probabilistic Methods for Algorithmic Discrete Mathematics, pp. 195–248 (1998)

Netflix.: Netflix prize. http://www.netflixprize.com (2006)

Rudin, C., Cortes, C., Mohri, M., Schapire, R.E.: Margin-Based Ranking Meets Boosting in the Middle. In: Auer, P., Meir, R. (eds.) COLT 2005. LNCS (LNAI), vol. 3559, pp. 63–78. Springer, Heidelberg (2005)

Shashua, A., Levin, A.: Ranking with large margin principle: Two approaches. Advances in Neural Information Processing Systems (NIPS 2003) (2003)

Vapnik, V.N.: Statistical learning theory. Wiley-Interscience, New York (1998)

Wahba, G.: Spline models for observational data. SIAM [Society for Industrial and Applied Mathematics] (1990)

Engineering Fast Route Planning Algorithms*

Peter Sanders and Dominik Schultes

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{sanders,schultes}@ira.uka.de

Abstract. Algorithms for route planning in transportation networks have recently undergone a rapid development, leading to methods that are up to one million times faster than Dijkstra's algorithm. We outline ideas, algorithms, implementations, and experimental methods behind this development. We also explain why the story is not over yet because dynamically changing networks, flexible objective functions, and new applications pose a lot of interesting challenges.

1 Introduction

Computing an optimal route in a transportation network between specified source and target nodes is one of the showpieces of real-world applications of algorithmics. We frequently use this functionality when planning trips with cars or public transportation. There are also many applications like logistic planning or traffic simulation that need to solve a huge number of shortest-path queries in transportation networks. In most of this paper we focus on the simplest case, a static *road* network with a fixed cost for each edge. The cost function may be any mix of travel time, distance, toll, energy consumption, scenic value, ... associated with the edges. Some of the techniques described below work best if the cost function is positively correlated with travel time. The task is to compute the costs of optimal paths between arbitrary source-target pairs. Some preprocessing is allowed but it has to be sufficiently fast and space efficient to scale to the road network of a continent.

The main part of this paper is Section 2, which explains the ideas behind several practically successful speedup techniques for static routing in road networks. Section 3 makes an attempt to summarize the development of performance over time. In Section 4 we outline generalizations for public transportation, time-dependent edge weights, outputting optimal paths, and dynamically changing networks. Section 5 describes some experiences we made with implementing route planning algorithms for large networks. Then, Section 6 explains our experimental approach giving some examples by applying it to the algorithms we implemented. We conclude in Section 7 with a discussion of some future challenges.

* Partially supported by DFG grant SA 933/1-3.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 23–36, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Static Routing in Large Road Networks

We consider a directed graph $G = (V, E)$ with n nodes and $m = \Theta(n)$ edges. An edge (u, v) has the nonnegative edge weight $w(u, v)$. A shortest-path query between a source node s and a target node t asks for the minimal weight $d(s, t)$ of any path from s to t. In static routing, the edge weights do not change so that it makes sense to perform some *precomputations*, store their results, and use this information to accelerate the queries. Obviously, there is some tradeoff between query time, preprocessing time, and space for preprocessed information. In particular, for large road networks it would be prohibitive to precompute and store shortest paths between all pairs of nodes.

2.1 "Classical Results"

Dijkstra's Algorithm [1]—the classical algorithm for route planning—maintains an array of *tentative distances* $D[u] \geq d(s, u)$ for each node. The algorithm *visits* (or *settles*) the nodes of the road network in the order of their distance to the source node and maintains the invariant that $D[u] = d(s, u)$ for visited nodes. We call the rank of node u in this order its *Dijkstra rank* $\mathrm{rk}_s(u)$. When a node u is visited, its outgoing edges (u, v) are *relaxed*, i.e., $D[v]$ is set to $\min(D[v], d(s, u) + w(u, v))$. Dijkstra's algorithm terminates when the target node is visited. The size of the search space is $O(n)$ and $n/2$ (nodes) on the average. We will assess the quality of route planning algorithms by looking at their *speedup* compared to Dijkstra's algorithm, i.e., how many times faster they can compute shortest-path distances.

Priority Queues. Dijkstra's algorithm can be implemented using $O(n)$ priority queue operations. In the comparison based model this leads to $O(n \log n)$ execution time. In other models of computation (e.g. [2]) and on the average [3], better bounds exist. However, in practice the impact of priority queues on performance for large road networks is rather limited since cache faults for accessing the graph are usually the main bottleneck. In addition, our experiments indicate that the impact of priority queue implementations diminishes with advanced speedup techniques since these techniques at the same time introduce additional overheads and dramatically reduce the queue sizes.

Bidirectional Search executes Dijkstra's algorithm simultaneously forward from the source and backwards from the target. Once some node has been visited from both directions, the shortest path can be derived from the information already gathered [4]. In a road network, where search spaces will take a roughly circular shape, we can expect a speedup around two —one disk with radius $d(s, t)$ has twice the area of two disks with half the radius. Bidirectional search is important since it can be combined with most other speedup techniques and, more importantly, because it is a necessary ingredient of the most efficient advanced techniques.

Geometric Goal Directed Search (A^*). The intuition behind goal directed search is that shortest paths 'should' lead in the general direction of the target. A^* search [5] achieves this by modifying the weight of edge (u, v) to $w(u, v) - \pi(u) + \pi(v)$ where $\pi(v)$ is a lower bound on $d(v, t)$. Note that this manipulation shortens edges that lead towards the target. Since the added and subtracted *vertex potentials* $\pi(v)$ cancel along any path, this modification of edge weights preserves shortest paths. Moreover, as long as all edge weights remain nonnegative, Dijkstra's algorithm can still be used. The classical way to use A^* for route planning in road maps estimates $d(v, t)$ based on the Euclidean distance between v and t and the average speed of the fastest road anywhere in the network. Since this is a very conservative estimation, the speedup for finding quickest routes is rather small. Goldberg et al. [6] even report a *slow-down* of more than a factor of two since the search space is not significantly reduced but a considerable overhead is added.

Heuristics. In the last decades, commercial navigation systems were developed which had to handle ever more detailed descriptions of road networks on rather low-powered processors. Vendors resolved to heuristics still used today that do not give any performance guarantees: use A^* search with estimates on $d(u, t)$ rather than lower bounds; do not look at 'unimportant' streets, unless you are close to the source or target [7]. The latter heuristic needs careful hand tuning of road classifications to produce reasonable results but yields considerable speedups.

2.2 Exploiting Hierarchy

Small Separators. Road networks are almost planar, i.e., most edges intersect only at nodes. Hence, techniques developed for planar graphs will often also work for road networks. Using $O(n \log^2 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [8,9] for directed planar graphs without negative cycles. Queries accurate within a factor $(1 + \epsilon)$ can be answered in near constant time using $O((n \log n)/\epsilon)$ space and preprocessing time [10]. Most of these theoretical approaches look difficult to use in practice since they are complicated and need superlinear space.

The first published practical approach to fast route planning [11] uses a set of nodes V_1 whose removal partitions the graph $G = G_0$ into small components. Now consider the *overlay graph* $G_1 = (V_1, E_1)$ where edges in E_1 are *shortcuts* corresponding to shortest paths in G that do not contain nodes from V_1 in their interior. Routing can now be restricted to G_1 and the components containing s and t respectively. This process can be iterated yielding a multi-level method. A limitation of this approach is that the graphs at higher levels become much more dense than the input graphs thus limiting the benefits gained from the hierarchy. Also, computing small separators and shortcuts can become quite costly for large graphs.

Reach-Based Routing. Let $R(v) := \max_{s,t\in V} R_{st}(v)$ denote the *reach* of node v where $R_{st}(v) := \min(d(s,v), d(v,t))$. Gutman [12] observed that a shortest-path search can be stopped at nodes with a reach too small to get to source or target from there. Variants of reach-based routing work with the reach of edges or characterize reach in terms of geometric distance rather than shortest-path distance. The first implementation had disappointing speedups (e.g. compared to [11]) and preprocessing times that would be prohibitive for large networks.

Highway Hierarchies. (HHs) [13,14] group nodes and edges in a hierarchy of levels by alternating between two procedures: Contraction (i.e., node reduction) removes low degree nodes by bypassing them with newly introduced shortcut edges. In particular, all nodes of degree one and two are removed by this process. Edge reduction removes *non-highway edges*, i.e., edges that only appear on shortest paths *close* to source or target. More specifically, every node v has a neighborhood radius $r(v)$ we are free to choose. An edge (u, v) is a highway edge if it belongs to some shortest path P from a node s to a node t such that (u, v) is neither fully contained in the neighborhood of s nor in the neighborhood of t, i.e., $d(s, v) > r(s)$ and $d(u, t) > r(t)$. In all our experiments, neighborhood radii are chosen such that each neighborhood contains a certain number H of nodes. H is a tuning parameter that can be used to control the rate at which the network shrinks. The query algorithm is very similar to bidirectional Dijkstra search with the difference that certain edges need not be expanded when the search is sufficiently far from source or target. HHs were the first speedup technique that could handle the largest available road networks giving query times measured in milliseconds. There are two main reasons for this success: Under the above contraction routines, the road network shrinks in a geometric fashion from level to level and remains sparse and near planar, i.e., levels of the HH are in some sense *self similar*. The other key property is that preprocessing can be done using limited local searches starting from each node. Preprocessing is also the most nontrivial aspect of HHs. In particular, long edges (e.g. long-distance ferry connections) make simple minded approaches far too slow. Instead we use fast heuristics that compute a superset of the set of highway edges.

Routing with HHs is similar to the heuristics used in commercial systems. The crucial difference is that HHs are guaranteed to find the optimal path. This qualitative improvement actually make HHs *much faster* than the heuristics. The latter have to make a precarious compromise between quality and size of the search space that relies on manual classification of the edges into levels of the hierarchy. In contrast, after setting a few quite robust tuning parameters, HH-preprocessing automatically computes a hierarchy aggressively tuned for high performance.

Advanced Reach-Based Routing. It turns out that the preprocessing techniques developed for HHs can be adapted to preprocessing reach information [15]. This makes reach computation faster and more accurate. More importantly, shortcuts make queries more effective by reducing the number of nodes traversed and by reducing the reach-values of the nodes bypassed by shortcuts.

Reach-based routing is slower than HHs both with respect to preprocessing time and query time. However, the latter can be improved by a combination with goal-directed search to a point where both methods have similar performance.

Highway-Node Routing. In [16] we generalize the multi-level routing scheme with overlay graphs so that it works with arbitrary sets of nodes rather than only with separators. This is achieved using a new query algorithm that stalls suboptimal branches of search on lower levels of the hierarchy. By using only *important* nodes for higher levels, we achieve query performance comparable to HHs. Preprocessing is done in two phases. In the first phase, nodes are classified into levels. We currently derive this information from a HH. In the second phase, we recursively compute the shortcuts bottom up. Shortcuts from level ℓ are found by local searches in level $\ell - 1$ starting from nodes in level ℓ. This second phase is very fast and easy to update when edge weights change.

Distance Tables. For HHs the network size shrinks geometrically from level to level. Once a level L has size $\Theta(\sqrt{n})$, we can afford to precompute and store a complete distance table for nodes in level L [14]. Using this table, we can stop a HH search when it has reached level L. To compute the shortest-path distance, it then suffices to lookup all shortest-path distances between nodes entering level L in forward and backward search respectively. Since the number of entrance nodes is not very large, one can achieve a speedup close to two compared to pure HH search.

Transit Node Routing precomputes not only a distance table for important (*transit*) nodes but also all relevant connections between the remaining nodes and the transit nodes [17,18]. Since it turns out that only about ten such *access connections* are needed per node, one can 'almost' reduce routing in large road networks to about 100 table lookups. Interestingly, the difficult queries are now the local ones where the shortest path does not touch any transit node. We solve this problem by introducing several *layers* of transit nodes. Between lower layer transit nodes, only those routes need to be stored that do not touch the higher layers. Transit node routing (e.g., using appropriate levels of a HH for transit node sets) reduces routing times to a few microseconds at the price of preprocessing times an order of magnitude larger than HHs alone.

2.3 Advanced Goal-Directed Search

Edge Labels. The idea behind edge labels is to precompute information for an edge e that specifies a set of nodes $M(e)$ with the property that $M(e)$ is a superset of all nodes that lie on a shortest path starting with e. In an s–t query, an edge e need not be relaxed if $t \notin M(e)$. In [11], $M(e)$ is specified by an *angular range*. More effective is information that can distinguish between long range and short range edges. In [19] many *geometric containers* are evaluated. Very good performance is observed for axis parallel rectangles. A disadvantage of geometric

containers is that they require a complete all-pairs shortest-path computation. Faster precomputation is possible by partitioning the graph into k regions that have similar size and only a small number of boundary nodes. Now $M(e)$ is represented as a k-vector of *edge flags* [20,21,22] where flag i indicates whether there is a shortest path containing e that leads to a node in region i. Edge flags can be computed using a single-source shortest-path computation from all boundary nodes of the regions. In [23] a faster variant of the preprocessing algorithm is introduced that takes advantage of the fact that for close boundary nodes the respective shortest-path trees are very similar.

Landmark A^*. Using the triangle inequality, quite strong bounds on shortest-path distances can be obtained by precomputing distances to a set of around 20 *landmark* nodes that are well distributed over the far ends of the network [6,24]. Using reasonable space and much less preprocessing time than for edge labels, these lower bounds yield considerable speedup for route planning.

Precomputed Cluster Distances (PCD). In [25], we give a different way to use precomputed distances for goal-directed search. We partition the network into clusters and then precompute the shortest connection between any pair of clusters U and V, i.e., $\min_{u \in U, v \in V} d(u, v)$. PCDs cannot be used together with A^* search since reduced edge weights can become negative. However, PCDs yield upper and lower bounds for distances that can be used to prune search. This gives speedup comparable to landmark-A^* using less space. Using the many-to-many routing techniques outlined in Section 4, cluster distances can also be computed efficiently.

2.4 Combinations

Bidirectional search can be profitably combined with almost all other speedup techniques. Indeed, it is a required ingredient of highway hierarchies, transit and highway-node routing and it gives more than the anticipated factor two for reach-based routing and edge flags. Willhalm et al. have made a systematic comparison of combinations of pre-2004 techniques [26,27]. Landmark A^* harmonizes very well with reach-based routing [15] whereas it gives only a small additional speedup when combined with HHs [28]. The reason is that in HHs, the search cannot be stopped when the search frontiers meet. However, the same approach is very effective at speeding up approximate shortest-path queries.

3 Chronological Summary—The Horse Race

In general it is difficult to compare speedup techniques even when restricting to road networks because there is a complex tradeoff between query time, preprocessing time and space consumption that depends on the network, on the objective function, and on the distribution of queries. Still, we believe that some ranking helps to compare the techniques. To keep things manageable, we will restrict ourselves to average query times for computing optimal travel times in one

Table 1. Chronological development of the fastest speedup techniques. As date for the first publication, we usually give the submission deadline of the respective conference. If available, we always selected measurements for the European road network even if they were conducted after the first publication. Otherwise, we *linearly* extrapolated the preprocessing times to the size of Europe, which can be seen as a *lower bound*. Note that not all speedup techniques have been preprocessed on the same machine.

method	first pub.	date mm/yy	data from	size $n/10^6$	space [B/node]	preproc. [min]	speedup
separator multi-level	[11]	04/99	[30]	0.1	?	$> 5\,400$	52
edge flags (basic)	[20]	03/04	[31]	6	13	299	523
landmark A^*	[6]	07/04	[32]	18	72	13	28
edge flags	[21,22]	01/05	[23]	1	141	2 163	1 470
HHs (basic)	[13]	04/05	[13]	18	29	161	2 645
reach + shortc. + A^*	[15]	10/05	[32]	18	82	1 625	1 559
	[32]	08/06	[32]	18	32	144	3 830
HHs	[14]	04/06	[14]	18	27	13	4 002
HHs + dist. tab. (mem)	[14]	04/06	[14]	18	17	55	4 582
HHs + dist. tab.	[14]	04/06	[14]	18	68	15	8 320
HHs + dist. tab. + A^*	[28]	08/06	[28]	18	76	22	11 496
high-perf. multi-level	[33]	06/06	[34]	18	181	11 520	401 109
transit nodes (eco)	[17]	10/06	[17]	18	110	46	471 881
transit nodes (gen)	[17]	10/06	[17]	18	251	164	1 129 143
highway nodes (mem)	[16]	01/07	[16]	18	2	24	4 079

of the largest networks that have been widely used—the road network of (Western) Europe provided by the company PTV AG and also used (in a slightly different version) in the 9th DIMACS Implementation Challenge [29]. We take the liberty to speculate on the performance of some older methods that have never been been run on such large graphs and whose actual implementations might fail when one would attempt it. In Tab. 1 we list speedup techniques in chronological order that are 'best' with respect to speedup for random queries and the largest networks tackled at that point. Sometimes we list variants with slower query times if they are considerably better with respect to space consumption or manageable graph size.

Before [11] the best method would have been a combination of bidirectional search with geometric A^* yielding speedups of 2–3 over unidirectional Dijkstra. The separator-based multi-level method from [11] can be expected to work even for large graphs if implemented carefully. Computing geometric containers [11,19] is still infeasible for large networks. Otherwise, they would achieve much larger speedups then the separator-based multi-level method. So far, computing edge flags has also been too expensive for Europe and the USA but speedups beyond 1 470 have been observed for a graph with one million nodes [23]. Landmark A^* works well for large graphs and achieves average speedup of 28 using reasonable space and preprocessing time [6]. The implementation of HHs [13] was the first that was able to handle Europe and the USA. This implementation wins over all previous methods in almost all aspects. A combination of reach-based routing

with landmark A^* [15] achieved better query times for the USA at the price of a considerably higher preprocessing time. At first, that code did not work well on the European network because it is difficult to handle the present long-distance ferry connections, but later it could be considerably improved [32]. By introducing distance tables and numerous other improvements, highway hierarchies took back the lead in query time [14] at the same time using an order of magnitude less preprocessing time than [13]. The cycle of innovation accelerated even further in 2006. Müller [33] aggressively precomputes the pieces of the search space needed for separator-based multi-level routing. At massive expense of space and preprocessing time, this method can achieve speedups around 400 000. (The original implementation cannot directly measure this because it has large overheads for disk access and parsing of XML-data). Independently, transit node routing was developed [17], that lifts the speedup to six orders of magnitude and completely replaces Dijkstra-like search by table lookups. Since transit node routing needs more space and preprocessing time than other methods, the story is not finished yet. For example, [16] achieves speedups comparable to HHs, using only a few bytes per node.

4 Generalizations

Many-to-Many Routing. In several applications we need complete distance tables between specified sets of source nodes S and target nodes T. For example, in logistics optimization, traffic simulation, and also within preprocessing techniques [25,17]. HHs (and other non-goal-directed bidirectional search methods [11,15,16]) can be adapted in such a way that only a single forward search from each source node and a single backward search from each target node is needed [35]. The basic idea is quite simple: Store the backward search spaces. Arrange them so that each node v stores an array of pairs of the form $(t, d(v,t))$ for all target nodes that have v in their backward search space. When a forward search from s settles a node v, these pairs are scanned and used to update the tentative distance from s to t. This is very efficient because the intersection between any two forward and backward search spaces is only around 100 for HHs and because scanning an array is much faster than priority queue operations and edge relaxations governing the cost of Dijkstra's algorithm. For example, for $|S| = |T| = 10\,000$, the implementation in [35] needs only one minute.

Outputting Paths. The efficiency of many speedup techniques stems from introducing shortcut edges and distance table entries that replace entire paths in the original graph [11,13,15,14,33,17]. A disadvantage of these approaches is that the search will output only a 'summary description' of the optimal path that involves shortcuts. Fortunately, it is quite straightforward to augment the shortcuts with information for unpacking them [28,35,17]. Since we can afford to precompute unpacked representations of the most frequently needed

long-distance shortcuts, outputting the path turns out to be up to four times *faster* than just traversing the edges in the original graph.

Flexible Objective Functions. The objective function in road networks depends in a complex way on the vehicle (fast? slow? too heavy for certain bridges?, ...) the behavior and goals of the driver (cost sensitive? thinks he is fast?, ...), the load, and many other aspects. While the appropriate edge weights can be computed from a few basic parameters, it is not feasible to perform preprocessing for all conceivable combinations. Currently, our best answer to this problem is highway-node routing [16]. Assuming that the important nodes are important for any reasonable objective function, only the second phase of preprocessing needs to be repeated. This is an order of magnitude faster than computing a HH.

Dynamization. In online car navigation, we want to take traffic jams etc. into account. On the first glance, this is the death blow for most speedup techniques since even a single traffic jam can invalidate any of the precomputed information. However, we can try to selectively update only the information affected by the traffic jam and/or relevant to the queries at hand. Several solutions are proposed at this conference. Landmark A^* can be dynamized either by noticing that lower bounds remain valid when edge weights can only increase, or by using known dynamic graph algorithms for updating the shortest-path trees from the landmarks [36]. We have developed highway-node routing for this purpose [16] because it allows fast and easy updates (2–40 ms per changed edge weight depending on the importance of the edge).

Public Transportation and Time-Dependent Edge Weights. The standard query in public transportation asks for the *earliest arrival* at the target node t given a departure time and the source node s. This means we are (explicitly or implicitly) searching in a time-dependent network where nodes are some point in space-time. This means that bidirectional search cannot be used out of the box since we do not know the target node in the time-dependent network. This is puzzling because the most successful schemes described above use bidirectional search. This leaves us with the choice to use the most effective unidirectional method, or to somehow make bidirectional search work. An obvious fix is to *guess* the arrival time. This can be done using binary search and there are many ways to tune this (e.g. by interpolation search).

Parallelization. Most preprocessing techniques and many-to-many routing are easy to parallelize [35]. Parallelizing the queries seems difficult and unnecessary (beyond executing forward and backward search in parallel) because the search spaces are already very small when using one of the best available techniques.

5 Implementation

Advanced algorithms for routing in road networks require thousands of lines of well written code and hence require considerable programming skill. In particular, it is not trivial to make the codes work for large networks. Here is an incomplete list of problems and complications that we have seen in routing projects: Graphs have to be glued together from several files. Tools for reading files crash for large graphs. Algorithm library code cannot handle large graphs at all. The code slows down by factor six when switching from a custom graph representation to an algorithm library. 32-bit code will not work. Libraries do not work with 64-bit code.

Our conclusion from these experiences was to design our own graph data structures adapted to the problem at hand. We use C++ with encapsulated abstract data types. Templates and inline functions make this possible without performance penalties.

Although speedup techniques developed by algorithmicists come with high level arguments why they should yield optimal paths, few come with a detailed correctness proof.[1] There are plenty of things that can go wrong both with the algorithms and their implementations. For example, we had several cases where the algorithm considered was only correct when all shortest paths are unique. The implementation can help here with extensive consistency checks in assertions and experiments that are always checked against naive implementations, i.e., queries are checked against Dijkstra's algorithm and fast preprocessing algorithms are checked against naive or old implementations. On the long run one also needs a flexible visualization tool that can draw pieces of large graphs, paths, search spaces, and node sets. Since we could not find tools for this purpose that scale to large road networks, we implemented our own system [37].

6 Experiments

Before 2005, speedup techniques were difficult to compare since studies were either based on very small graphs or on proprietary data that could not be used by other groups. In particular, for 'newcomers' it was almost impossible to start working in this field. In [13] we were able to obtain two large road networks for the subcontinents Western Europe and the USA. The European network was made available for scientific use by the company PTV AG. We extracted the USA network from publicly available geographical data [38]. Since then, variants of these graphs have been used for most studies. We view it as likely that the sudden availability of data and the fast rate of innovation since then are not a coincidence. These networks are not directly annotated with edge weights but with lengths and road categories. By setting average speeds for each road category one can obtain realistic estimates of travel time.

Another important issue are which queries should be measured. The obvious choice is to use randomly selected node pairs on the largest available graph.

[1] We are working on one for HHs.

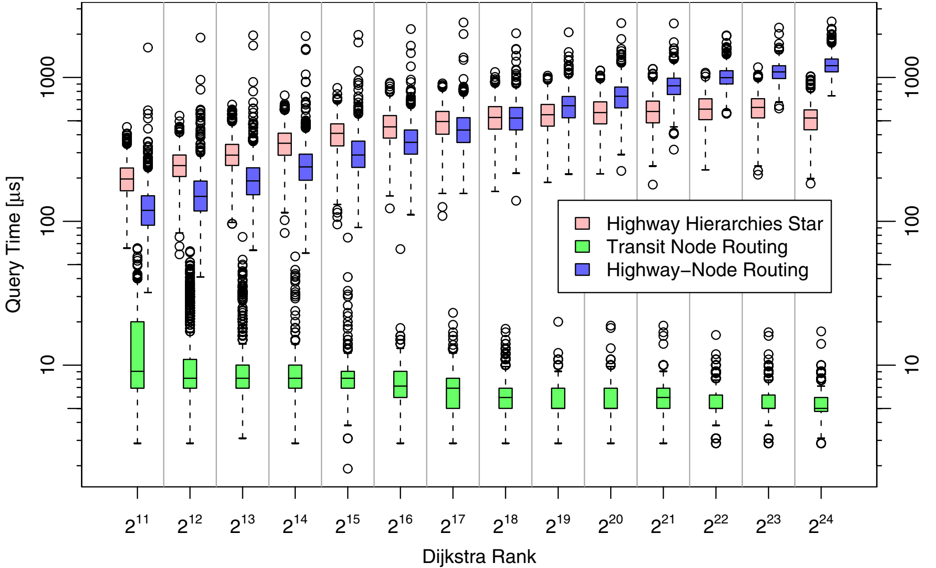


Fig. 1. Query performance of various speedup techniques against Dijkstra rank. The median speedups of HHs and highway-node routing cross at $r = 2^{18}$ since HHs are augmented with distance tables and goal direction. These techniques are particularly effective for large r and could also be adapted to highway-node routing.

Although this is a meaningful number, it is not quite satisfactory since most queries will produce very long paths (thousands of kilometers) that are actually rare in practice. Other studies therefore use random queries on a variety of subgraphs. However, this leads to a plethora of arbitrary choices that make it difficult to compare results. In particular, authors will always be tempted to choose only those subgraphs for which their method performs well.

Sets of real world queries would certainly be interesting, but so far we do not have them and it is also unlikely that a sample taken from one server is actually representative for the entire spectrum of route planning applications. We therefore chose a more systematic approach [13] that has also been adopted in several other studies: We generate a random query with a specified 'locality' r by choosing a random starting node s, and a target node t with Dijkstra rank $\mathrm{rk}_s(t) = r$ (i.e., the r-th node visited by a Dijkstra search from s). In our studies, we generate many such queries for each r which is a power of two. We then plot the distribution with median, quartiles, and outliers for each of these values of r. For the European road network, Fig. 1 shows the results for highway hierarchies combined with a distance table and goal-directed search, transit node routing, and highway-node routing. We view it as quite important to give information on the entire distribution since some speedup techniques have large fluctuations in query time.

In some cases, e.g., for HHs, it is also possible to compute good upper bounds on the search space size of *all* queries that can ever happen for a given graph [14]. We view this as a quite good surrogate for the absence of meaningful worst case upper bounds that would apply to all conceivable networks.

7 Conclusions and Open Problems

Speedup techniques for routing in static road networks have made tremendous progress in the last few years. Were it not for challenging applications such as logistics planning and traffic simulation, we could even say that the methods available now are *too* fast since other overheads like displaying routes or transmitting them over the network are the bottleneck once the query time is below a few milliseconds.

A major challenge is to close the gap to theory, e.g., by giving meaningful characterizations of 'well-behaved' networks that allow provably good worst-case bounds. In particular, we would like to know for which networks the existing techniques will also work, e.g., for communication networks, VLSI design, social networks, computer games, graphs derived from geometric routing problems, ...

Even routing techniques themselves are not quite finished yet. For example, we can look at better ways to select transit and highway nodes. We could also try to integrate edge labels with hierarchical routing schemes so that hierarchies help to approximate edge labels that in turn allow strong goal direction for queries.

Perhaps the main academic challenge is to go beyond static point-to-point routing. Public transportation and road networks with *time-dependent* travel times are an obvious generalization that should also be combined with updates of edge weights due to traffic jams. Further beyond that, we want multi-criteria optimization for individual paths and we want to compute social optima and Nash-equilibria taking the entire traffic in an area into account.

The main practical issue is how to transfer the academic results into applications. Many details have to be taken care of, like turn penalties, implementations on mobile devices, user specific objective functions, and compatibility with existing parts of the applications. The difficulty here is not so much scientific but one of finding the right approach to cooperation between academia and industry.

Acknowledgements

We would like to thank our coauthors on route planning, Holger Bast, Daniel Delling, Stefan Funke, Sebastian Knopp, Domagoj Matijevic, Jens Maue, Frank Schulz, and Dorothea Wagner for their valuable contributions. We also had many interesting discussions with Rob van den Berg, Sebastian Egner, Andrew Goldberg, Joaquim Gromicho, Martin Holzer, Stefan Hug, Ali Nowbakht Irani, Ekkehard Köhler, Ulrich Lauther, Ramon Lentink, Rolf Möhring, Kirill Müller, Matthias Müller-Hannemann, Paul Perdon, Heiko Schilling, Mikkel Thorup, Jaques Verrier, Peter Vortisch, Renato Werneck, and Thomas Willhalm.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
2. Thorup, M.: Integer priority queues with decrease key in constant time and the single source shortest paths problem. In: 35th ACM Symposium on Theory of Computing. pp. 149–158 (2003)
3. Meyer, U.: Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In: 12th Symposium on Discrete Algorithms. pp. 797–806 (2001)
4. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, Princeton (1962)
5. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on System Science and Cybernetics 4(2), 100–107 (1968)
6. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms, pp. 156–165. ACM Press, New York (2005)
7. Ishikawa, K., Ogawa, M., Azume, S., Ito, T.: Map Navigation Software of the Electro Multivision of the '91 Toyota Soarer. In: IEEE Int. Conf. Vehicle Navig. Inform. Syst. pp. 463–473 (1991)
8. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci. 72(5), 868–889 (2006)
9. Klein, P.: Multiple-source shortest paths in planar graphs. In: 16th ACM-SIAM Symposium on Discrete Algorithms, SIAM, pp. 146–155 (2005)
10. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. In: 42nd IEEE Symposium on Foundations of Computer Science. pp. 242–251 (2001)
11. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. In: Vitter, J.S., Zaroliagis, C.D. (eds.) WAE 1999. LNCS, vol. 1668, pp. 110–123. Springer, Heidelberg (1999)
12. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments. pp. 100–111 (2004)
13. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 568–579. Springer, Heidelberg (2005)
14. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)
15. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments, Miami (2006) 129–143
16. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: 6th Workshop on Experimental Algorithms (2007)
17. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In: transit to constant time shortest-path queries in road networks. In: Workshop on Algorithm Engineering and Experiments (2007)
18. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks with transit nodes. Science (to appear, 2007)
19. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 776–787. Springer, Heidelberg (2003)

20. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In: Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung. Vol. 22, pp. 219–230, IfGI prints, Institut für Geoinformatik, Münster (2004)
21. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: 4th International Workshop on Efficient and Experimental Algorithms (2005)
22. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: 4th International Workshop on Efficient and Experimental Algorithms. pp. 189–202 (2005)
23. Köhler, E., Möhring, R.H., Schilling, H.: Fast point-to-point shortest path computations with arc-flags. In: 9th DIMACS Implementation Challenge [29] (2006)
24. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Workshop on Algorithm Engineering and Experimentation. pp. 26–40 (2005)
25. Maue, J., Sanders, P., Matijevic, D.: Goal directed shortest path queries using Precomputed Cluster Distances. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 316–328. Springer, Heidelberg (2006)
26. Holzer, M., Schulz, F., Willhalm, T.: Combining speed-up techniques for shortest-path computations. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 269–284. Springer, Heidelberg (2004)
27. Willhalm, T.: Engineering Shortest Path and Layout Algorithms for Large Graphs. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik (2005)
28. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge [29] (2006)
29. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9 (2006)
30. Holzer, M., Schulz, F., Wagner, D.: Engineering multi-level overlay graphs for shortest-path queries. invited for ACM Journal of Experimental Algorithmics (special issue Alenex 2006) (2007)
31. Lauther, U.: An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In: 9th DIMACS Implementation Challenge [29] (2006)
32. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Better landmarks within reach. In: 9th DIMACS Implementation Challenge [29] (2006)
33. Müller, K.: Design and implementation of an efficient hierarchical speed-up technique for computation of exact shortest paths in graphs. Master's thesis, Universtät Karlsruhe supervised by Delling, D., Holzer, M., Schulz, F., Wagner, D.: (2006)
34. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-performance multi-level graphs. In: 9th DIMACS Implementation Challenge [29] (2006)
35. Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D.: Computing many-to-many shortest paths using highway hierarchies. In: Workshop on Algorithm Engineering and Experiments (2007)
36. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: 6th Workshop on Experimental Algorithms (2007)
37. Bingmann, T.: Visualisierung sehr großer Graphen. Student Research Project, Universität Karlsruhe, supervised by Sanders, P., Schultes, D. (2006)
38. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html (2002)

Random Models for Geometric Graphs
(Abstract)

Maria Serna

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Jordi Girona 1-3, Ω Building, E-08034 Barcelona, Spain
mjserna@lsi.upc.edu

Abstract. In the last decade there has been an increasing interest in graphs whose nodes are placed in the plane. In particular when modeling the communication pattern in wireless ad-hoc networks. The different communication ways or protocol implementations have directed the interest of the commnity to study and use different intersection graph families as basic models of communication. In this talk we review those models when the graph nodes are placed at random in the plane. In general we assume that the set of vertices is a random set of points generated by placing n points uniformly at random in the unit square. The basic distance model, the *random geometric graph* connects two points if they are at distance at most r where r is a parameter of the model [2]. A second model is the *k-neighbor graph* in which each node selects as neighbors the k-nearest neighbors in P [3]. Another variation, inspired by the communication pattern of directional radio frequency and optical networks, is the *random sector graph*, a generalization of the random geometric graph introduced in [1]. In the setting under consideration, each node has a fixed angle α ($0 < \alpha \leq 2\pi$) defining a sector S_i of transmission determined by a random angle between the sector and the horizontal axis. Every node that falls inside of S_i can potentially receive the signal emitted by i. In this talk we survey the main properties and parameters of such random graph families, and their use in the modelling and experimental evaluation of algorithms and protocols for several network problems.

References

1. Díaz, J., Petit, J., Serna, M.: A random graph model for optical networks of sensors. IEEE Transactions on Mobile Computing 2, 143–154 (2003)
2. Penrose, M.D.: Random Geometric Graphs. Oxford Studies in Probability, Oxford U.P (2003)
3. Xue, F., Kumar, P.R.: The number of neighbors needed for connectivity of wireless networks. Wireless Networks 10(2), 169–181 (2004)

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, p. 37, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Better Landmarks Within Reach

Andrew V. Goldberg[1], Haim Kaplan[2,⋆], and Renato F. Werneck[1]

[1] Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA
{goldberg,renatow}@microsoft.com
[2] School of Mathematical Sciences, Tel Aviv University, Israel
haimk@post.tau.ac.il.

Abstract. We present significant improvements to a practical algorithm for the point-to-point shortest path problem on road networks that combines A^* search, landmark-based lower bounds, and reach-based pruning. Through reach-aware landmarks, better use of cache, and improved algorithms for reach computation, we make preprocessing and queries faster while reducing the overall space requirements. On the road networks of the USA or Europe, the shortest path between two random vertices can be found in about one millisecond after one or two hours of preprocessing. The algorithm is also effective on two-dimensional grids.

1 Introduction

We study the *point-to-point shortest path problem* (P2P): given a directed graph $G = (V, A)$ with nonnegative arc lengths, a source s, and a destination t, find a shortest path from s to t. Preprocessing is allowed, as long as the amount of precomputed data is linear in the input graph size; preprocessing time is limited by practical considerations. The algorithms have two components: a *preprocessing algorithm* that computes auxiliary data and a *query algorithm* that computes an answer for a given s-t pair. We are interested in exact solutions only.

No nontrivial theoretical results are known for the general P2P problem. For the special case of undirected planar graphs, sublinear bounds are known [7]. Experimental work on exact algorithms with preprocessing includes [8,10,11,12, 14,16,18,19,20,22,23,25]. Next we discuss the most relevant recent developments.

Gutman [12] introduced the notion of *vertex reach*. Informally, the reach of a vertex v is large if v is close to the middle of some long shortest path and small otherwise. Intuitively, local intersections have low reach and highways have high reach. Gutman showed how to prune an s-t search based on (upper bounds on) reaches and (lower bounds on) vertex distances from s and to t, using Euclidean distances as lower bounds. He also observed that efficiency improves further when reaches are combined with Euclidean-based A^* search, which uses lower bounds on the distance to the destination to direct the search towards it.

Goldberg and Harrelson [8] (see also [11]) have shown that A^* search (without reaches) performs significantly better when landmark-based lower bounds

⋆ Work partially done while this author was visiting Microsoft Research Silicon Valley.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 38–51, 2007.
© Springer-Verlag Berlin Heidelberg 2007

are used instead of Euclidean ones. The preprocessing algorithm computes and stores the distances between every vertex and a small set of special vertices, the landmarks. Queries use the triangle inequality to obtain lower bounds on the distances between any two vertices in the graph. This leads to the ALT (A^* search, landmarks, and triangle inequality) algorithm for the P2P problem.

Sanders and Schultes [19,20] use the notion of *highway hierarchies* to design efficient algorithms for road networks. The preprocessing algorithm builds a hierarchy of increasingly sparse *highway networks*; queries start at the original graph and gradually move to upper levels of the hierarchy, greatly reducing the search space. To magnify the natural hierarchy of road networks, the algorithm adds *shortcuts* to the graph: additional edges with length equal to the original shortest path between their endpoints.

We have recently shown [10] how shortcuts significantly improve the performance of reach-based algorithms, in terms of both preprocessing and queries. The resulting algorithm, called RE, can be combined with ALT in a natural way, leading to the REAL algorithm. Section 2 presents a more detailed overview of these algorithms. This paper continues our study of reach-based point-to-point shortest paths algorithms and their combination with landmark-based A^* search.

An important observation about RE and REAL is that, unless s and t are very close to each other, an s-t search will visit mostly vertices of high reach. Therefore, as shown in Section 3.1, reordering vertices by reach can significantly improve the locality (and running times) of reach-based queries. In Section 3.2 we develop the concept of *reach-aware landmarks*, based on the intuition that accurate lower bounds are more important for vertices of high reach. In fact, as we suggested in [10], one can keep landmark data for these vertices only. Balancing the number of landmarks and the fraction of distances that is actually kept, a memory-time trade-off is established. For large graphs, we were able to reduce both time and space requirements compared to our previous algorithm.

In addition, motivated by the work of Sanders and Schultes [20], Section 3.4 describes how shortcuts can be used to bypass vertices of arbitrary (but usually low) degree, instead of just degree-two vertices as our previous method did. This improves both preprocessing and query performance. We also study two techniques for accelerating reach computation: a modified version of our partial trees algorithm (for finding approximate reaches) and a novel algorithm for finding exact reaches. They are described in Sections 3.3 and 3.5, respectively.

Experimental results are presented in Section 4. On road networks, the algorithmic improvements lead to substantial savings in time, especially for preprocessing, and memory. The road networks of Europe or the USA can be preprocessed in one or two hours, and queries take about one millisecond (while Dijkstra's algorithm takes seconds). We also obtain reasonably good results for 2-dimensional grids. For grids of higher dimension and for random graphs, however, preprocessing becomes too expensive and query speedups are only marginal.

A preliminary version of this paper [9] was presented at the 9th DIMACS Implementation Challenge: Shortest Paths [5], where several other algorithms were introduced [1,3,4,15,17,21]. Section 5 discusses these recent developments.

2 Algorithm Overview

Our algorithms have four main components: Dijkstra's algorithm, reach-based pruning, A^* search, and their combination. We discuss each in turn.

Dijkstra's algorithm. The *labeling method* finds shortest paths from a source s to all vertices in the graph as follows (see e.g. [24]). It keeps for every vertex v its distance label $d(v)$, parent $p(v)$, and status $S(v) \in \{\texttt{unreached}, \texttt{labeled}, \texttt{scanned}\}$. Initially, $d(v) = \infty$, $p(v) = nil$, and $S(v) = \texttt{unreached}$ for every vertex v. The method starts by setting $d(s) = 0$ and $S(s) = \texttt{labeled}$. While there are labeled vertices, it picks a labeled vertex v, *relaxes* all arcs out of v, and sets $S(v) = \texttt{scanned}$. To relax an arc (v, w), one checks if $d(w) > d(v) + \ell(v, w)$ and, if true, sets $d(w) = d(v) + \ell(v, w)$, $p(w) = v$, and $S(w) = \texttt{labeled}$. By always scanning the vertex with the smallest label, *Dijkstra's algorithm* [6] ensures that no vertex is scanned more than once. The P2P version can stop when it is about to scan the target t: the s-t path defined by the parent pointers is the solution.

One can also run Dijkstra's algorithm from the target on the reverse graph to find the shortest t-s path. The *bidirectional algorithm* alternates between the forward and reverse searches, each maintaining its own set of distance labels (denoted by $d_f(\cdot)$ and $d_r(\cdot)$, respectively). When an arc (v, w) is relaxed by the forward search and w has already been scanned by the reverse search, we know the shortest s-v and w-t paths have lengths $d_f(v)$ and $d_r(w)$, respectively. If $d_f(v) + \ell(v, w) + d_r(w)$ is less than the shortest path distance found so far (initially ∞), we update it. We perform similar updates during the reverse search. The algorithm stops when the two searches meet.

Reach-based pruning. Given a path P from s to t and a vertex v on P, the *reach of v with respect to P* is the minimum of the lengths of the s-v and v-t subpaths of P. The *reach* of v, $r(v)$, is the maximum, over all **shortest** paths P through v, of the reach of v with respect to P [12]. We assume shortest paths are unique, which can be achieved through perturbation. Computing exact reaches is impractical for large graphs; we must resort to upper bounds instead. We denote an upper bound on $r(v)$ by $\overline{r}(v)$, and a lower bound on the distance $\mathrm{dist}(v, w)$ from v to w by $\underline{\mathrm{dist}}(v, w)$. If, during an s-t query, we observe that $\overline{r}(v) < \underline{\mathrm{dist}}(s, v)$ and $\overline{r}(v) < \underline{\mathrm{dist}}(v, t)$, then v is not on a shortest path from s to t and therefore Dijkstra's algorithm can prune the search at v.

To apply this, we need lower bounds on $\mathrm{dist}(s, v)$ and $\mathrm{dist}(v, t)$. During a bidirectional search, one can use the bounds implicit in the search itself [10]. Consider the forward direction (the reverse case is similar), and let γ be the smallest distance label of a labeled vertex in the reverse direction (i.e., the top-most label in the reverse heap). If a vertex v has not been scanned in the reverse direction, then γ is a lower bound on the distance from v to t. We can prune the search at v if v has not been scanned in the reverse direction, $\bar{r}(v) < d_f(v)$, and $\bar{r}(v) < \gamma$. The algorithm can still stop when the searches meet.

Reach upper bounds are computed during the preprocessing phase. It works in rounds, each associated with a threshold ϵ_i (which grows exponentially with i). Round i bounds the reach of vertices whose reach is at most ϵ_i. It does so by growing

partial trees from each vertex, each with depth roughly $2\epsilon_i$. A vertex whose reach is bounded is eliminated from the graph and considered only indirectly in subsequent rounds. We also add *shortcuts* to the graph before each round [10]. Given two edges (u, v) and (v, w), a shortcut is a new edge (u, w) with length $\ell(u, v) + \ell(v, w)$ ($\ell(\cdot, \cdot)$ denotes the length of an edge). When ties are broken appropriately, the shortcut will ensure that v does not belong to the shortest path between u and w, potentially reducing v's reach, thus making pruning more effective and speeding up preprocessing. After all reaches are bounded, a *refinement step* builds the graph induced by the $\lceil 5\sqrt{n} \rceil$ vertices with highest reach bounds and recomputes the reaches using an exact algorithm. The preprocessing algorithm, as well as improvements relative to [10], are discussed in detail in Sections 3.3, 3.4, and 3.5.

A^* search and the ALT *algorithm.* A *potential function* maps vertices to reals. Given a potential function π, the *reduced cost* of an arc is defined as $\ell_\pi(v, w) = \ell(v, w) - \pi(v) + \pi(w)$. Suppose we replace ℓ by ℓ_π. The length of every s-t path changes by the same amount, $\pi(t) - \pi(s)$, so finding shortest paths in the original graph is equivalent to finding shortest paths in the transformed graph. Let π_f be a potential function such that $\pi_f(v)$ gives an estimate on the distance from v to t. In the context of this paper, *A^* search* [13] is an algorithm that works like Dijkstra's algorithm, but at each step selects a labeled vertex v with the smallest *key*, defined as $k_f(v) = d_f(v) + \pi_f(v)$, to scan next. This effectively guides the search towards t. It is easy to see that A^* search is equivalent to Dijkstra's algorithm on the graph with length function ℓ_{π_f}. If π_f is such that ℓ_π is nonnegative for all arcs (i.e., if π_f is *feasible*), the algorithm will find the correct shortest paths. We use as potential functions lower bounds on the distance from v to the target t. During the preprocessing stage, we pick a constant number of vertices as *landmarks* and store distances between them and every vertex in the graph; queries use these distances, together with the triangle inequality, to compute the lower bounds. The ALT algorithm is a bidirectional version of A^* search with landmark bounds and triangle inequality.

Combining reaches and landmarks. We can combine A^* search and reaches in the obvious way: running A^* search and pruning vertices based on reach conditions. Specifically, when A^* search is about to scan a vertex v with key $k_f(v) = d_f(v) + \pi_f(v)$, it can prune the search at v if $\bar{r}(v) < d_f(v)$ and $\bar{r}(v) < \pi_f(v)$. Note that this method (which we call REAL) has two preprocessing algorithms: reach computation and landmark generation. Although they are in principle independent, Section 3.2 will show that it might be useful to take reaches into account when generating landmarks.

3 Algorithmic Improvements

3.1 Improving Locality

When reaches are available, a typical point-to-point query spends most of its time scanning high-reach vertices. Except at the very beginning of the search, low-reach vertices are pruned by reach. This suggests an obvious optimization:

during preprocessing, reorder the vertices such that high-reach vertices are close together in memory to improve cache locality. Simply sorting vertices in non-increasing order of reach would destroy the locality of the input, which is often quite high. Instead, we partition the vertices into equal-sized sets, the first with the $n/2$ higher-reach vertices, and the other with the rest. We keep the original relative ordering in each part, then recursively process the first (high-reach) part. This also facilitates the optimizations described below.

3.2 Reach-Aware Landmarks

We can reduce the memory requirements of the algorithm by storing landmark distances only for high-reach vertices, with marginal performance degradation. If we use the saved space to add more landmarks, we get a wide range of trade-offs between query performance and memory requirement. We call the resulting method, a variant of REAL, the *partial landmark algorithm*.

Queries for this algorithm work as follows. Let R, the *reach threshold*, be the smallest value such that *all* vertices with reach at least R have landmark distances available. We say these vertices have *high reach*. Queries start as RE, with reach pruning but without A^* search, until both balls searched have radius R (or the algorithm terminates). From this point on, only vertices with reach R or higher will be scanned. We switch to REAL by removing labeled vertices from the heaps and reinserting them with new keys that incorporate lower bounds.

To process a vertex v, A^* search needs lower bounds on the distance from v to t (in the forward search) or from s to v (in the reverse search). They are computed with the triangle inequality, which requires distances between these vertices (v, s, and t) and the landmarks. These are guaranteed to be available for v, which has high reach, but not for s or t; for them, we must use *proxies*. The proxy for s, which we denote by s', is the high-reach vertex that is closest to s (t is treated similarly). A lower bound on $\text{dist}(s, v)$ using distances *to* a landmark L is given by $\text{dist}(s, v) \geq \text{dist}(s', L) - \text{dist}(v, L) - \text{dist}(s', s)$. Other bounds (using distances from landmarks and involving s and t) can be computed in a similar way. Proxies (and related distances) are computed during the initialization phase of the query algorithm with a multiple-source version of Dijkstra's algorithm.

We use the *avoid* algorithm [11] to select landmarks: it picks landmarks one by one, always in regions of the graph not already "covered" by existing landmarks. It does so by assigning to each vertex a *weight* that measures how well-covered it is. We changed the algorithm slightly from [11]: instead of computing the weights of all vertices in the graph, we only consider n/k of them (where k is the number of landmarks). This makes the algorithm linear in k, instead of quadratic, without a significant effect on solution quality.

3.3 Growing Partial Trees

Next we describe the partial-trees routine executed in each iteration of our preprocessing algorithm. For simplicity, we describe the algorithm as if it computed vertex reaches; it actually computes *arc reaches* and eventually converts them

to vertex reaches [10]. (The reach of an arc (v, w) with respect to a shortest s-t path containing it is $\min\{\text{dist}(s, w), \text{dist}(v, t)\}$.) In each round, we are given a graph $G = (V, A)$ and a threshold ϵ, and our goal is to find upper bounds on the reaches of vertices in V whose actual reaches are smaller than ϵ. The remaining reaches will be bounded in subsequent rounds, when ϵ will be larger.

Fix a vertex v. To prove that $r(v) < \epsilon$, we must consider all shortest paths through v, but only *minimal paths* must be processed explicitly. Let $P_{st} = (s, s', \ldots, v, \ldots, t', t)$ be the shortest s-t path, and assume that v has reach at least ϵ with respect to this path. Path P_{st} is *ϵ-minimal* with respect to v if and only if the reaches of v with respect to $P_{s't}$ and $P_{st'}$ are both smaller than ϵ.

The algorithm works by growing a *partial tree* T_r from each vertex $r \in V$. It runs Dijkstra's algorithm from r, but stops as soon as it can prove that all ϵ-minimal paths starting at r have been considered. Let v be a vertex in this tree, and let x be the first vertex (besides r) on the path from r to v. We say that v is an *inner vertex* if either (1) $v = r$ or (2) $d(x, v) < \epsilon$, where $d(x, v)$ denotes the distance between x and v in the tree. The inner vertices are those whose reaches we will try to bound. When v is not an inner vertex, no path P_{rw} starting at r will be ϵ-minimal with respect to v: if v's reach is greater than ϵ with respect to P_{rw}, it will also be greater than ϵ with respect to P_{xw}. To get accurate bounds on reaches, we must grow the tree until every inner vertex v has one of two properties: (1) v has no labeled (unscanned) descendents; or (2) v has at least one scanned descendent whose distance to v is ϵ or greater. For efficiency, we relax the second condition and stop when every labeled vertex is within distance greater than ϵ from the closest inner vertex.

Once the tree is built, we process it. Given an inner vertex v, we know its *depth*, equal to $d(r, v)$. In $O(|T_r|)$ time, we can also compute its *height*, defined as the distance from v to its farthest descendent (labeled or scanned). The reach of v with respect to T_r is the minimum between its depth and its height, and the reach of v with respect to the entire graph is the maximum over all such reaches. If this maximum is ϵ or greater, we declare the reach to be ∞.

Penalties. As described, the algorithm assumes that partial trees will be grown from every vertex in the graph. We would like, however, to run the partial-trees routine even after some of the vertices have been eliminated in a previous iteration. We use *penalties* to account for the eliminated vertices. The *in-penalty* of a vertex v is the maximum over the reaches of all arcs (u, v) that have already been eliminated; *out-penalties* are defined similarly, considering outgoing arcs instead. Partial trees are processed as before, but the definitions of *height* and *depth* must change. The (redefined) depth of a vertex v within a tree T_r, denoted by $depth_r(v)$, is the distance from r to v plus the in-penalty of r. The (modified) height of v is the distance to its farthest descendent not in T_r itself, but in a *pseudo-tree* in which each vertex v in T_r is attached to a *pseudo-leaf* v' by an arc whose length is equal to the out-penalty of v.

For correctness, it is enough to take penalties into account only when processing the tree, as we did in [10]. We can, however, use penalties to stop growing partial trees sooner, thus speeding up preprocessing. First, we now consider

vertex v to be an inner vertex if either (1) $v = r$ or (2) $depth_x(v) < \epsilon$ (recall that x is the second vertex on the path from r to v). If $v \neq r$ and $depth_r(v) <$ *in-penalty*(v), however, v will not be considered an inner vertex (because its modified depth will be even higher in the tree rooted at v), and neither will its descendents. Second, we stop growing the tree when no labeled (unscanned) vertex is *relevant*. All inner vertices are considered relevant; an outer vertex v is relevant if $d(u, v)+$*out-penalty*$(v) \leq \epsilon$, where u is the closest inner ancestor of v.

3.4 Adding Shortcuts

Our previous implementation of the preprocessing procedure [10] only shortcuts vertices with degree two. Sanders and Schultes [20] suggested shortcutting other vertices of small degree as well, which is more general and works better for both preprocessing and queries. A vertex v can be *bypassed* as follows. First, we examine all pairs of incoming/outgoing arcs $((u, v), (v, w))$ with $u \neq w$. For each pair, if the arc (u, w) is not in the graph, we add a shortcut arc (u, w) of length $\ell(u, v) + \ell(v, w)$. Otherwise, we set $\ell(u, w) \leftarrow \min\{\ell(u, w), \ell(u, v) + \ell(v, w)\}$. Finally, we delete v and all arcs adjacent to it from the current graph.

The processing algorithm will produce a graph containing all original arcs and all shortcuts. To prevent the graph from being too large, we only bypass v if both its in-degree and its out-degree are bounded by a constant (5 in our experiments); this ensures that only $O(n)$ arcs will be added. In addition, we consider the ratio c_v between the number of new arcs added and the number of arcs deleted by the procedure above. A vertex v is deemed bypassable only if $c_v \leq c$, where c is a user-defined parameter. For road networks, we used $c = 0.5$ in the first round of preprocessing, 1.0 in the second, and 1.5 in the remaining rounds. As a result, relatively few shortcuts are added during the first two rounds, when the graph is larger but shrinks faster. For two- and three-dimensional grids, which do not shrink as fast, we fixed c at 1.0 and 2.0, respectively.

We also consider two additional measures (besides c_v) related to v: the length of the longest shortcut arc introduced when v is bypassed, and the largest reach of an arc adjacent to v (which will be removed). The maximum between these two values is the *cost* of v, and it must be bounded by $\epsilon_i/2$ during iteration i for the vertex to be considered bypassable. As explained in [10], long arcs and large penalties can decrease the quality of the reach upper bounds provided by the preprocessing algorithm; they should not appear too soon. When deciding which vertex to bypass next, we take those that minimize the product between c_v and cost, since they are less likely to affect the bypassability of their neighbors.

3.5 Exact Reach Computation

The standard algorithm for computing exact reaches (during the refinement step) builds a shortest path tree from each vertex r in the graph and computes the minimum between the depth and the height of each vertex v in the tree. The maximum of these minima over all trees will be the reach of v. We developed an algorithm that has the same worst-case complexity, but can be significantly

faster in practice on road networks. It follows the same basic principle, but builds parts of some of the shortest path trees *implicitly* by reusing previously found subtrees. The algorithm partitions the vertices of the graph into k regions. (In our experiments, we picked the regions of the Voronoi diagram of $k = \sqrt{n}$ randomly selected vertices.) The *frontier* of a region A is the set of vertices $v \in A$ such that there exists at least one arc (v, w) with $w \neq A$. When processing a region, the algorithm first grows full shortest path trees from the frontier. Typically, they will have large subtrees in common, and it is easy to see that the same subtrees would appear in the full shortest path trees rooted at non-frontier vertices as well. It therefore suffices to grow *truncated* trees from these vertices, which account for the common subtrees only implicitly.

4 Experimental Results

Our code was written in C++ and compiled with Microsoft Visual C++ 2005. All tests were performed on a dual-processor, 2.4 GHz AMD Opteron running Microsoft Windows Server 2003 with 16 GB of RAM, 32 KB instruction and 32 KB data level 1 cache per processor, and 2 MB of level 2 cache. Our code is single-threaded, but an execution is not pinned to a single processor.

We tested the ALT (with 16 landmarks), RE, REAL-$(16, 1)$, and REAL-$(64, 16)$ algorithms. Here REAL-(i, j) denotes an algorithm that uses i landmarks but maintains landmark data for n/j highest-reach vertices only (when $j = 1$, all landmark distances are kept, as in the original REAL algorithm). Due to space constraints, we omit detailed results for other values of i and j. In general, however, we observed that moderately increasing sparsity does affect running times, but not too much: on large road networks, REAL-(16,16) is less than 30% slower than REAL-(16,1), for instance. The sparser the landmarks, the more the algorithm will rely on RE (at the beginning); the slight increase in the number of vertices scanned is offset by the fact that scans are on average faster, since RE does not need to access landmark data.

For machine calibration purposes, we also ran the DIMACS Challenge implementation of the P2P version of Dijkstra's algorithm, denoted by D, on the largest road networks. Our experiments on road networks are described in Section 4.1, and experiments on grid graphs are reported in Section 4.2.

4.1 Road Networks

We first test our algorithm on the road networks of the USA and Europe, which belong to the DIMACS Challenge [5] data set. The USA is symmetric and has 23 947 347 vertices and 58 333 444 arcs; Europe is directed, with 18 010 173 vertices and 42 560 279 arcs. Both graphs are strongly connected. Two length functions are available in each case: travel times and travel distances.

Random queries. For each graph and each length function, we tested the algorithms on 1 000 pairs of vertices picked uniformly at random. Table 1 reports the average query time (in milliseconds), the average number of scanned vertices

Table 1. Data for random queries on Europe and USA graphs

GRAPH	METHOD	PREP. TIME (min)	DISK SPACE (MB)	QUERY AVG SC.	MAX SC.	TIME (ms)
Europe (times)	ALT	13.2	1597	82348	993015	160.34
	RE	82.7	626	4643	8989	3.47
	REAL-(16,1)	96.8	1849	814	4709	1.22
	REAL-(64,16)	140.8	1015	679	2955	1.11
	RE-OLD	1558	570	16122	34118	13.5
	REAL-OLD	1625	1793	1867	8499	2.8
	HH	15	1570	884	—	0.8
	HH-mem	55	692	1976	—	1.4
	D	—	393	8984289	—	4365.81
USA (times)	ALT	18.6	2563	187968	2183718	400.51
	RE	44.3	890	2317	4735	1.81
	REAL-(16,1)	63.9	3028	675	3011	1.14
	REAL-(64,16)	121.0	1575	540	1937	1.05
	RE-OLD	366	830	3851	8722	4.50
	REAL-OLD	459	2392	891	3667	1.84
	HH	18	1686	1076	—	0.88
	HH-mem	65	919	2217	—	1.60
	D	—	536	11808864	—	5440.49
Europe (distances)	ALT	10.1	1622	240750	3306755	430.02
	RE	49.3	664	7045	12958	5.53
	REAL-(16,1)	60.3	1913	882	5973	1.52
	REAL-(64,16)	89.8	1066	583	2774	1.16
	D	—	393	8991955	—	2934.24
USA (distances)	ALT	14.5	2417	276195	2910133	530.35
	RE	70.8	928	7104	13706	5.97
	REAL-(16,1)	87.8	2932	892	4894	1.80
	REAL-(64,16)	138.1	1585	628	4076	1.48
	D	—	536	11782104	—	4576.02

and, when available, the maximum number of scanned vertices. Also shown are the total preprocessing time and the total space on disk used by the preprocessed data. For D, this is the graph itself; for ALT, this includes the graph and landmark data; for RE, it includes the graph with shortcuts and an array of vertex reaches; the data for REAL includes the data for RE plus landmark data.

For travel times, the table also reports the performance of other algorithms available at the time of writing. We give the data for our previous implementations, RE-OLD and REAL-OLD from [10] (run on the same machine). REAL-OLD uses 16 landmarks selected with the *maxcover* method (which finds slightly better landmarks than *avoid*, but is slower). In addition, we present results for the highway hierarchy-based algorithm of Sanders and Schultes from [20], run sequentially on a dual-core 2.0 GHz AMD Opteron machine (which is about 20% faster than our machine on the DIMACS benchmark due to a different memory architecture). There are two versions of their algorithm: HH-mem, entirely based

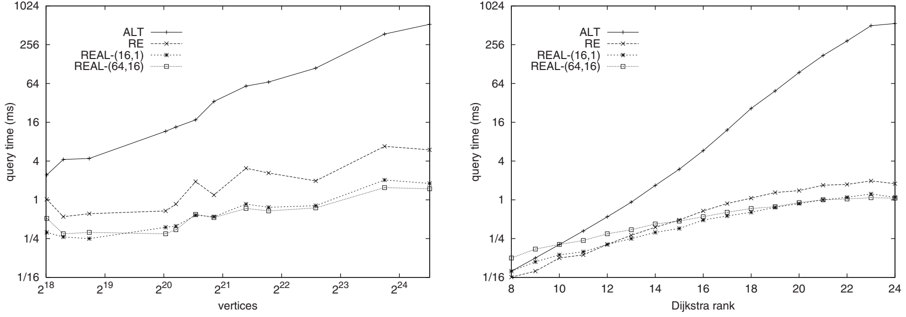


Fig. 1. USA queries: random on subgraphs (left) and local on the full graph (right)

on highway hierarchies, and HH, which replaces high levels of the hierarchy by a table with distances between all pairs of vertices in the corresponding graph.

The table shows that RE is considerably faster than ALT for queries, and that REAL-(16,1) yields an additional speedup. Comparing REAL-(16,1) to REAL-(64,16), we see they have almost identical query performance with transit times, and that REAL-(64,16) is slightly better with travel distances. Given that REAL-(64,16) requires about half as much disk space, it has the edge for these queries. With travel distances, REAL-(64,16) wins both in time and in space, but preprocessing takes roughly twice as long. RE is less robust than REAL.

With travel times, RE and REAL substantially improve on their old counterparts, especially in terms of preprocessing time. RE and HH-mem have similar performance on USA, and HH-mem is slightly better on Europe. For queries, REAL-(64,16) performs similarly to HH: REAL-(64,16) uses less space, while HH is slightly faster. Preprocessing for HH, however, is faster, especially for Europe.

Graph size dependence. To test the performance on smaller graphs, we performed random queries on the subgraphs of USA that are part of the DIMACS data set; their sizes range from 264 thousand (NYC) to 14 million (CTR) vertices. Figure 1 (left) shows how query times scale with graph size when travel times are used as the length function. Although times tend to increase with graph size, they are not strictly monotone: graph structure clearly plays a part. Reach-based algorithms have better asymptotic performance than ALT. Regarding preprocessing (not shown in the figure), with a fixed number of landmarks ALT is roughly linear in the graph size. With 16 landmarks, ALT preprocessing is faster than RE, and the ratio between the two remains roughly constant as the graph size increases; with 64, reach computation and landmark selection take roughly the same time.

Local queries. Up to this point, we have considered only random queries; we now test what happens when queries are more local. If v is k-th vertex scanned when Dijkstra's algorithm is run from s, then the *Dijkstra rank* of v with respect to s is $\lfloor \log_2 k \rfloor$ (our definition differs slightly from [19]). To generate a local query with rank k, we pick s uniformly at random from V and pick t uniformly at random from all vertices with Dijkstra rank k with respect to s. Figure 1 (right) shows the average query times as a function of Dijkstra rank (1 000 pairs were tested in each case). For

Table 2. Effect of applying (•) or discarding (◦) each of the improvements to RE on USA with travel times: generalized shortcuts (S), penalty-aware partial trees (P), faster exact reach computation (E) and improved locality (L)

FEATURES				PREP. TIME	DISK SPACE	QUERY		
S	P	E	L	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
•	•	•	•	44.3	890	2317	4735	1.81
◦	•	•	•	63.8	817	3861	7679	2.64
•	◦	•	•	76.2	888	2272	4633	1.75
•	•	◦	•	58.7	890	2317	4735	1.81
•	•	•	◦	44.3	890	2317	4735	3.20
◦	◦	◦	◦	156.1	816	3741	7388	3.89

high Dijkstra ranks, the results are similar to those for random queries: ALT is the slowest algorithm, and the REAL variants are the fastest. For small Dijkstra ranks, REAL-(16,1) scans the fewest vertices, but due to higher overhead its running time is slightly worse than that of RE. REAL-(64,16) mostly visits low-reach vertices and thus fails to take advantage of the landmark data. It scans about the same number of vertices as RE, but is slower due to higher overhead. Although ALT has the worst asymptotic performance, for small ranks it scans only slightly more vertices than RE. As the rank grows, REAL-(64,16) eventually catches up with REAL-(16,1).

Improvement breakdown. Table 1 has shown that the new version of RE is significantly more efficient than RE-OLD. Table 2 shows how each of the four major improvements affect the performance of RE on USA with travel times. Starting with all improvements, we turn them off one at a time, and then all at once. Preprocessing is accelerated by generalized shortcuts (Section 3.4), penalty-aware partial trees (Section 3.3), and faster exact reach computation (Section 3.5). Sorting by reach to improve locality (Section 3.1) actually slows preprocessing, but by a negligible amount. When these improvements are combined, the overall speedup is more than 3.5. On Europe with travel times (not shown), the combined speedup is more than 6. Queries benefit from generalized shortcuts and sorting by reach, and are largely unaffected by the other improvements.

Refinement step. During preprocessing, the refinement step recomputes the $\lceil 5\sqrt{n} \rceil$ highest reaches (24 469 vertices in the USA graph) with an exact algorithm. If we quadruple this value, RE query times decrease from 1.86 ms to 1.66 ms (with travel times as lengths); however, preprocessing time increases from under 45 minutes to 2.5 hours. With no refinement step, the preprocessing time decreases to 32 minutes, but query times increase to 2.00 ms. Recomputing all reaches is too expensive. Even on Bay Area, which has only 321 270 vertices, exact reach computation takes almost 2.5 hours with the new algorithm (10 hours with the standard one). Computing upper bounds takes less than a minute, but queries are 40% faster with exact reaches.

Retrieving the shortest path. The query times reported so far for RE and REAL consider only finding the shortest path on the graph with shortcuts. This path has much fewer arcs then the corresponding path in the original graph. On

Table 3. Average results for 1 000 random queries on 2-dimensional grids

		PREP. TIME	DISK SPACE	QUERY		
VERTICES	METHOD	(min)	(MB)	AVG SC.	MAX SC.	TIME (ms)
65536	ALT	0.05	11.5	851	6563	1.19
	RE	1.83	3.4	2195	3706	1.50
	REAL-(16,1)	1.87	12.7	214	1108	0.30
	REAL-(64,16)	2.00	5.9	1999	3134	1.69
	D	—	2.2	33752	—	14.83
131044	ALT	0.11	23.8	1404	11535	3.62
	RE	4.72	6.8	3045	5025	2.33
	REAL-(16,1)	4.82	26.1	266	1261	0.61
	REAL-(64,16)	5.09	12.1	2370	3513	2.19
	D	—	4.5	66865	—	30.72
262144	ALT	0.22	48.3	2439	27936	5.72
	RE	16.61	13.2	4384	6933	3.52
	REAL-(16,1)	16.83	52.5	410	2024	0.88
	REAL-(64,16)	17.42	23.9	1869	2453	2.28
	D	—	9.0	134492	—	63.58
524176	ALT	0.27	96.6	6057	65664	8.11
	RE	15.45	25.9	6334	9843	4.23
	REAL-(16,1)	15.76	104.5	524	2663	1.08
	REAL-(64,16)	16.68	47.5	2001	3113	1.97
	D	—	18.0	275589	—	112.80

USA with travel times, for example, the shortest path between a random pair of vertices has around 5 000 vertices in the original graph, but only about 30 in the graph with shortcuts. We can retrieve the original path in time proportional to its length, which means about 1 ms on the USA graph. This is comparable to the time it takes for REAL to compute the distances. For applications that require a full description of the path, our algorithms are therefore close to optimal.

4.2 Grid Graphs

Reach-based pruning works well on road networks because they have a natural highway hierarchy, so that relatively few vertices have high reach. We also tested the algorithms on graphs without an obvious hierarchy. We created square 2-dimensional with the `spgrid` generator, available at the 9th DIMACS Challenge download page. The graphs are directed and each vertex is connected to its neighbors in the grid with arcs of length chosen uniformly at random from the range $[1, n]$, where n is the number of vertices. Comparing the results in Table 3 to those reported in [10], we see that our new preprocessing algorithm is an order of magnitude faster on these graphs. Query times improve by a factor of about five. This makes RE competitive with ALT; in fact, RE appears to be asymptotically faster. Performance of REAL-(16,1) improves as well. This shows that reaches help even when a graph does not have an obvious highway hierarchy, and that the applicability of REAL is not restricted to road networks. On the

largest grid, it is four times faster than ALT, and two orders of magnitude faster than Dijkstra's algorithm. On such small graphs, extra landmarks do not help much, and REAL-(64,16) does not perform as well as REAL-(16,1).

Similar experiments on cube-shaped three-dimensional grids (not shown) revealed that the ALT algorithm is much less effective than on two-dimensional grids. For a quarter of a million vertices, queries are only five times as fast as bidirectional Dijkstra's algorithm. A combination with pruning by reach does improve the algorithm for large graphs, but only marginally. Moreover, reach computation becomes asymptotically slower (the time roughly triples when the graph size doubles), thus making preprocessing large graphs prohibitively expensive. Results for higher-dimension grids and random graphs were even worse.

5 Final Remarks

Several other papers presented at the 9th DIMACS Implementation Challenge [5] also dealt with the P2P problem. Lauther [17] and Köhler et al. [15] presented algorithms based on arc flags, but their (preprocessing and query) running times are dominated by REAL and HH. Delling, Sanders, et al. [4] presented a variant of the partial landmarks algorithm in the context of highway hierarchies, but with only modest speedups; for technical reasons A^* search cannot be combined naturally with HH. Delling, Holzer, et al. [3] showed how multi-level graphs can support random queries in less than 1 ms, but only after weeks of preprocessing.

The best results were those based on *transit node routing*, introduced by Bast et al. [1] and combined with highway hierarchies by Sanders and Schultes [21] (see also [2]). With travel times, the road networks of both USA and Europe can be processed in about three hours and random queries take 5 μs on average. With travel distances, preprocessing takes about eight hours, and average query times are close to 0.1 ms. Performance would probably be worse on grids.

Queries with transit node routing are significantly faster than with REAL. Our method does appear to be more robust, however, when the length function changes. Moreover, these approaches are not mutually exclusive. As Bast et al. observe [2], reaches could be used instead of highway hierarchies to compute the transit nodes and the corresponding distance tables. An actual implementation of the combined algorithm is an interesting topic for future research.

References

1. Bast, H., Funke, S., Matijevic, D.: TRANSIT: Ultrafast shortest-path queries with linear-time preprocessing. 9th DIMACS Implementation Challenge (2006)
2. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In transit to constant time shortest-path queries in road networks. In: Proc. 9th ALENEX. SIAM, 2007. Available at http://www.mpi-inf.mpg.de/~bast/tmp/transit.pdf
3. Delling, D., Holzer, M., Muller, K., Schulz, F., Wagner, D.: High-performance multi-level graphs. In: 9th DIMACS Implementation Challenge (2006)
4. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge (2006)

5. Demetrescu, C., Goldberg, A.V., Johnson, D.S.: 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9/ (2006)
6. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numer. Math. 1, 269–271 (1959)
7. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. In: Proc. 42nd FOCS, pp. 232–241 (2001)
8. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A* Search Meets Graph Theory. In: Proc. 16th SODA, pp. 156–165 (2005)
9. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Better landmarks within reach. 9th DIMACS Implementation Challenge (2006)
10. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In: Proc. 8th ALENEX. SIAM (2006)
11. Goldberg, A.V., Werneck, R.F.: Computing Point-to-Point Shortest Paths from External Memory. In: Proc. 7th ALENEX, SIAM pp. 26–40(2005)
12. Gutman, R.: Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In: Proc. 6th ALENEX, pp. 100–111 (2004)
13. Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on System Science and Cybernetics, vol. SSC-4(2) (1968)
14. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 126–138. Springer, Heidelberg (2005)
15. Köhler, E., Möhring, R.H., Schilling, H.: Fast point-to-point shortest path computations with arc-flags. In: 9th DIMACS Implementation Challenge (2006)
16. Lauther, U.: An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In: IfGIprints 22, Institut fuer Geoinformatik, Universitaet Muenster (ISBN 3-936616-22-1), pp. 219–230 (2004)
17. Lauther, U.: An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In: 9th DIMACS Implementation Challenge (2006)
18. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: Nikoletseas, S.E. (ed.) WEA (2005). LNCS, vol. 3503, pp. 189–202. Springer, Heidelberg (2005)
19. Sanders, P., Schultes, D.: Fast and Exact Shortest Path Queries Using Highway Hierarchies. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 568–579. Springer, Heidelberg (2005)
20. Sanders, P., Schultes, D.: Engineering Highway Hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)
21. Sanders, P., Schultes, D.: Robust, almost constant time shortest-path queries on road networks. In: 9th DIMACS Implementation Challenge (2006)
22. Schultes, D.: Fast and Exact Shortest Path Queries Using Highway Hierarchies. Master's thesis, Department of Computer Science, Universität des Saarlandes, Germany (2005)
23. Schulz, F., Wagner, D., Weihe, K.: Using Multi-Level Graphs for Timetable Information. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 43–59. Springer, Heidelberg (2002)
24. Tarjan, R.E.: Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA (1983)
25. Wagner, D., Willhalm, T.: Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 776–787. Springer, Heidelberg (2003)

Landmark-Based Routing in Dynamic Graphs*

Daniel Delling and Dorothea Wagner

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{delling,wagner}@ira.uka.de

Abstract. Many speed-up techniques for route planning in static graphs exist, only few of them are proven to work in a dynamic scenario. Most of them use preprocessed information, which has to be updated whenever the graph is changed. However, goal directed search based on landmarks (ALT) still performs correct queries as long as an edge weight does not drop below its initial value. In this work, we evaluate the robustness of ALT with respect to traffic jams. It turns out that—by increasing the efficiency of ALT—we are able to perform fast (down to 20 ms on the Western European network) random queries in a dynamic scenario without updating the preprocessing as long as the changes in the network are moderate. Furthermore, we present how to update the preprocessed data without any additional space consumption and how to adapt the ALT algorithm to a time-dependent scenario. A time-dependent scenario models predictable changes in the network, e.g. traffic jams due to rush hour.

1 Introduction

Computing shortest paths in graphs $G = (V, E)$ is used in many real-world applications like route planning in road networks, timetable information for railways, or scheduling for airplanes. In general, Dijkstra's algorithm [1] finds the shortest path between a given source s and target t. Unfortunately, the algorithm is far too slow to be used on huge datasets. Thus, several speed-up techniques exist [2] yielding faster query times for typical instances, e.g., road or railway networks. Recent research [3,4] even made the calculation of the distance between two points in road networks of the size of Europe a matter of microseconds. Thus, at least for road networks, shortest path computation seems to be solved.

However, most of the existing techniques require a *static* graph, i.e. the graph is known in advance and does not change between two shortest path computations. A more realistic scenario is a dynamic one: new roads are constructed and closed or traffic jams occur. Furthermore, we often know in advance that the motorways are crowded during rush hour. In this work, we adapt the known technique of goal directed search based on landmarks—called ALT [5]—to these dynamic scenarios.

* Partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 52–65, 2007.
© Springer-Verlag Berlin Heidelberg 2007

1.1 Related Work

We focus on related work on dynamic speed-up techniques. For static scenarios, see [2] for an overview. An adaption of DIJKSTRA's algorithm to a scenario where travel times depend on the daytime, the *time-dependent* scenario, can be found in [6]. Throughout the paper, we distinguish time-dependent and *time-independent* scenarios. For the latter, edge weights are not dependent on the daytime. A classical speed-up technique is *bidirectional* DIJKSTRA which also starts a search from the target. As bidirectional DIJKSTRA uses no preprocessing, it can be used in a time-independent dynamic scenario without any effort. However, its adaption to a time-dependent scenario is more complicated as the arrival time is unknown in such a scenario.

Goal directed search, also called A^* [7], pushes the search towards a target by adding a potential to the priority of each node. The usage of Euclidean potentials requires no preprocessing. The ALT algorithm, introduced in [5], obtains the potential from the distances to certain landmarks in the graph. Although this approach requires a preprocessing step, it is superior with respect to search space and query times. Goldberg and Harrelson state that ALT may work well in a dynamic scenario. In this work, we persue and advance their ideas. In [8], A^* using Euclidean potentials is adapted to a time-dependent scenario.

Geometric containers [9] attach a label to each edge that represents all nodes to which a shortest path starts with this particular edge. A dynamization has been published in [9] yielding suboptimal containers if edge weights decrease. In [10], ideas from highway hierarchies [11] and overlay graphs [2] are combined yielding very good query times in dynamic road networks.

Closely related to dynamic shortest path computation are the Single-Source and All-Pair-Shortest-Path problems. Both of them have been studied in a dynamic scenario [12,13].

1.2 Overview

In Section 2 we review the ALT algorithm, introduced in [14] and enhanced in [15]. We improve the original algorithm by storing landmark data more efficiently. In Section 3, we briefly discuss how to model traffic in predictable and unexpected cases. The adaption of ALT to our models from Section 3 is located in Section 4. First, we show how to update the preprocessing efficiently *without* any additional requirements of data. The update is based on dynamic shortest path trees that can be reconstructed from the graph with data provided by ALT. However, as already mentioned in [14], it turns out that for the most common type of update, i.e., traffic jams, the update of the preprocessing needs not be done in order to keep queries correct. Finally, we are able to adapt a unidirectional variant of ALT to the time-dependent model. An extensive experimental evaluation can be found in Section 5, proving the feasibility of our approach. There, we focus on the performance of ALT with no preprocessing updates when traffic jams occur. Section 6 concludes this work by a summary and possible future work.

2 Goaldirected Search Based on Landmarks

In this section, we explain the known ALT algorithm [14]. In general, the algorithm is a variant of bidirectional A^* search [7] in combination with landmarks. We follow the implementation presented in [15] enriched by implementation details that increase efficiency.

The search space of DIJKSTRA's algorithm can be interpreted as a circle around the source. By adding a 'good' potential $\pi : V \rightarrow \mathbb{R}$ to the priority of each node, the order in which nodes are removed from the priority queue is altered in such a way that nodes lying on a shortest path to the target yield a low priority. In [7], it is shown that this technique—known as A^*—is equivalent to DIJKSTRA's algorithm on a graph with *reduced costs*, formally $w_\pi(u, v) = w(u, v) - \pi(u) + \pi(v)$. Since DIJKSTRA's algorithm works only on nonnegative edge costs, not all potentials are allowed. We call a potential π *feasible* if $w_\pi(u, v) \geq 0$ for all $(u, v) \in E$. The distance from each node v of G to the target t is the distance from v to t in the graph with reduced edge costs minus the potential of t plus the potential of v. So, if the potential $\pi(t)$ of the target t is zero, $\pi(v)$ provides a *lower bound* for the distance from v to the target t. There exist several techniques [16] to obtain feasible potentials using the layout of a graph. The ALT algorithm uses a small number of nodes—so called *landmarks*—and the triangle inequality to compute feasible potentials. Given a set $S \subseteq V$ of landmarks and distances $d(L, v), d(v, L)$ for all nodes $v \in V$ and landmarks $L \in S$, the following triangle inequations hold: $d(u, v) + d(v, L) \geq d(u, L)$ and $d(L, u) + d(u, v) \geq d(L, v)$. Therefore, $\underline{d}(u, t) := \max_{L \in S} \max\{d(u, L) - d(t, L), d(L, t) - d(L, u)\}$ provides a lower bound for the distance $d(u, t)$ and, thus, can be used as a potential for u.

The quality of the lower bounds highly depends on the quality of the selected landmarks. Thus, several selection strategies exist. To this point, no technique is known for picking landmarks that yield the smallest search space for random queries. Thus, several heuristics exist; the best are *avoid* and *maxCover* [15].

As already mentioned, ALT is a bidirectional variant of A^*. In general, the combination of A^* and bidirectional search is not that easy as it seems. Correctness can only be guaranteed if π_f—the potential for the forward search—and π_r—the potential for the backward search—are *consistent*. This means $w_{\pi_f}(u, v)$ in G is equal to $w_{\pi_r}(v, u)$ in the reverse graph. We use the variant of an average potential function [7] defined as $p_f(v) = (\pi_f(v) - \pi_r(v))/2$ for the forward and $p_r(v) = (\pi_r(v) - \pi_f(v))/2 = -p_f(v)$ for the backward search. Note, that π_f provides better potentials than p_f. Moreover, for a bidirectional variant, the stopping criterion has to be altered: Stop the search if the sum of minimum keys in the forward and the backward queue exceeds $\mu + p_f(s)$, where μ represents the tentative shortest path length.

Improved Efficiency. One downside of ALT seemed to be its low efficiency. In [17], a reduction of factor 44 in search space only leads to a reduction in query times of factor 21. By storing landmark data more efficiently, this gap can be reduced. First, we sort the landmarks by ID in ascending order. The

distances from and to a landmark are stored as one 64-bit integer for each node and landmark: The upper 32 bits refer to the 'to' distance and the lower to the 'from' distance. Thus, we initialize a 64-bit vector of size $|S| \cdot |V|$. Both distances of node number $i \in [0, |V| - 1]$ and landmark number $j \in [0, |S| - 1]$ are stored at position $|S| \cdot i + j$. As a consequence, when computing the potential for given node n, we only have one access to the main memory in most times.

3 Modeling Traffic

In the following, we briefly discuss how to model several scenarios of updates due to traffic in road networks. We cover unexpected and predictable updates.

Dynamic Updates. The most common updates of transit times in road networks are those of *traffic jams*. This can just be slight increases for many roads due to rush hour. Nevertheless, increases of higher percentage can happen as well. In the worst case, routes may be closed completely. Currently, traffic reports concentrate on motorways. Nevertheless, with new technologies like car-to-car communication [18] information for all roads will be available. This will lead to scenarios where updates happen quite frequently for all kinds of roads. Analyzing the type of updates one may notice that transit times may increase (high traffic) and may decrease afterwards but will not drop below the transit times of an 'empty' road. Overall, a dynamic speed-up technique for shortest path computation should handle this kind of updates very well and very fast. In the ideal case, the technique should not need any update at all.

Like traffic jams, *construction sites* increase transit times when they are installed and decrease them when the work is done. In most cases, the transit times do not fall below the original value. So, for this case, an updating routine for handling traffic jams can also handle construction sites.

Another type of change in the structure of a network is the construction or demolition of roads. This can be modeled by insertions or deletions of edges and/or nodes. On the one hand, these type of updates are known in advance and on the hand, they happen not very often. Thus, a dynamic algorithm should be capable of handling these updates but the performance on these updates is not that important like traffic jams.

Normally, transit times of roads are calculated by the average speed on these roads. But profiles of clients differ: some want to be as fast as possible, others want to save fuel. Furthermore, the type of vehicle has an impact on travel times (a truck is slower than a sports car) and even worse, some roads are closed to specific types of vehicles.

Time-Dependency. Most of the changes in transit times are predictable. We know in advance that during rush hour, transit times are higher than at night. And we know that the motorways are crowded at the beginning and end of a holiday. This scenario can be modelled by a *time-dependent* graph [8] which assigns several weights to a specific edge. Each weight represents the travel time

at a certain time. As a consequence, the result of an s-t query depends on the time of departure from s. In [19], it is shown that time-dependent routing gets more complicated if the graph is not time consistent. Time consistency means that for each edge (u, v) a delayed departure from u does *not* yield a earlier arrival in v. Throughout this paper, we focus on time consistent graphs. Note that our models from Section 3 can be used in a time-dependent scenario by updating some or all of the weights assigned to an edge.

4 Dynamization

In this section, we discuss how the preprocessing of the ALT algorithm can be updated efficiently. Furthermore, we discuss a variant where the preprocessing has to be updated only very few times. However, this approach may lead to a loss in performance. At the end of this section we introduce a time-dependent variant of the unidirectional ALT algorithm.

4.1 Updating the Preprocessing

The preprocessing of ALT consists of two steps: the landmark selection and calculating the distance labels. As the selection of landmarks are heuristics, we settle for *static* landmarks, i.e., we do not reposition landmarks if the graph is altered. The update of the distance labels can be realized by dynamic shortest path trees. For each landmark, we store two trees: one for the forward edges, one for the backward edges. Whenever an edge is altered we update the tree structure including the distance labels of the nodes. In the following we discuss a memory efficient implementation of dynamic shortest path trees. The construction of a tree can be done by running a complete Dijkstra from the root of the tree.

Updating Shortest Path Trees. In [12] the update of a shortest path tree is discussed. The approach is based on a modified Dijkstra, trying to identify the regions that have to be updated after a change in edge weight. Therefore, a tree data structure is used in order to retrieve all successors and the parent of a node quickly. As road graphs are sparse we do not need to store any additional information to implement these operations. The successors of n can be determined by checking for each target t of all outgoing edges e whether $d(n) + w(e) = d(t)$ holds. If it holds, t can be interpreted as successor of n. Analogously, we are able to determine the parent of a node n: Iterate all sources s of the incoming edges e of n. If $d(s)+w(e) = d(n)$ holds, s is the parent of n. This implementation allows to iterate all successors of n in $O(\delta)$ where δ is the degree of n. The parent of n can be found in $O(\delta)$ as well. Note that we may obtain a different tree structure than rerunning a complete Dijkstra, but as we are only interested in distance labels, this approach is sufficient for the correctness of ALT.

The advantage of this approach is memory consumption. Keeping all distance labels for 16 landmarks on the road network of Western Europe in memory already requires about 2.2 GB of RAM (32 trees with 18 million nodes, 32 bit

per node). Every additional pointer would need an additional amount of 2.2 GB. Thus, more advanced tree structures (1, 2, or 4 pointers) lead to an overhead that does not seem worth the effort.

4.2 Two Variants of the Dynamic ALT Algorithm

Eager Dynamic ALT. In the previous section we explained how to update the preprocessed data of ALT. Thus, we could use the update routine whenever the graph is altered. In the following, we call this variant of the dynamic ALT the *eager dynamic* version.

Lazy Dynamic ALT. However, analyzing our dynamic scenarios from Section 3 and the ALT algorithm from Section 2 we observe two important facts. On the one hand, ALT-queries only lose correctness if the potential of an edge results in a negative edge cost in the reduced graph. This can only happen if the cost of the edge drops below the value during preprocessing. On the other hand, for the most common update type—traffic jams—edge weights may increase and decrease but do not drop below the initial value of empty roads. Thus, a potential computed on the graph without any traffic stays feasible for those kinds of updates even when not updating the distances from and to all landmarks. Due to this observation, we may do the preprocessing for empty roads and use the obtained potentials even though an edge is perturbed. In [14], this idea was stated to be semi-dynamic, allowing only increases in edge weights. Nevertheless, as our update routine does not need any additional information, we are able to handle all kinds of updates. Our *lazy dynamic* variant of ALT leaves the preprocessing untouched unless the cost of an edge drops below its initial value.

This approach may lead to an increase in search space. If an edge e on the shortest path is increased without updating the preprocessing, the weight of e is also increased in G', the graph with reduced costs. Thus, the length of the shortest path increases in G'. So, the search stops later because more nodes are inserted in the priority queue (cf. the stopping criterion in Section 2). However, as long as the edges are not on the shortest path of a requested query the search space does not increase. More precisely, the search space may even decrease because nodes 'behind' the updated edge are inserted later into the priority queue.

4.3 The Time-Dependent ALT Algorithm

In time-independent scenarios, ALT is implemented as bidirectional search. But in time-dependent scenarios, a backward search is prohibited. Thus, we have to use an unidirectional variant of ALT that only performs a forward search. As a consequence, we may use the known stopping criterion of DIJKSTRA's algorithm: Stop the search when the target node is taken from the priority queue. Furthermore, we may use the potential π_f instead of the average potential p_f yielding better lower bounds (Section 2).

Based on the ideas from 4.2, we can adapt the unidirectional ALT algorithm to the time-dependent scenario. When doing the preprocessing, we use the minimum weight of each edge to compute the distance labels. It is obvious that we obtain a feasible potential. The time-dependent ALT algorithm works analogously to an unidirectional ALT but calculates the estimated departure time from a node in order to obtain the correct edge weight. We alter the priority of each node by adding the potential computed during preprocessing.

Note that the time-dependent ALT algorithm also works in a dynamic time-dependent scenario. Using the same arguments from Section 4.2, the algorithm still performs accurate queries as long as an edge weight does not drop below the value used during the preprocessing. If this happens, the distance labels can be updated using the routine from Section 4.1.

5 Experiments

Our experimental evaluation was done on one CPU of a dual AMD Opteron 252 running SUSE Linux 10.1. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.1, using optimization level 3.

As inputs, we use the road map of Western Europe, provided by PTV AG for scientific use, and the US network taken from the TIGER/Line Files. The former graph has approximately 18 million nodes and 42.6 million edges, where edge lengths correspond to travel times. The corresponding figures for the USA are 23.9 million and 58.3 million, respectively. Each edge belongs to one of four main categories representing motorways, national roads, local streets, and urban streets. The European network has a fifth category representing rural roads. For updates, we do not consider these rural roads. In general, we measure a *low perturbation* of an edge by an increase of its weight by factor 2. For a *high perturbation* we use an increase by factor 10. In the following, we identify the unidirectional, bidirectional and time-dependent ALT algorithm by *uni*-ALT, ALT and *time*-ALT, respectively. The number of landmarks is indicated by a number after the algorithm, e.g. ALT-16 for 16 landmarks.

The Static ALT Algorithm. In order to compare our ALT implementations in a dynamic scenario, we report the performance of the bi- and unidirectional ALT algorithm in a static scenario. We evaluate different numbers of landmarks with respect to preprocessing, search space and query times performing 10 000 uniformly distributed random s-t queries. Due to memory requirements we used *avoid* for selecting 32 and 64 landmarks. For less landmarks, we used the superior *maxCover* heuristic. Table 1 gives an overview.

We see for bidirectional ALT that doubling the number of landmarks reduces search space and query times by factor 2, which does not hold for the unidirectional variant. This is due to the fact that goal direction works best on motorways as these roads mostly have reduced costs of 0 in the reduced graph. In the unidirectional search, one has to leave to motorway in order to reach the target. This drawback cannot be compensated by more landmarks.

Table 1. Preprocessing, search space, and query times of uni- and bidirectional ALT and Dijkstra based on 10 000 random *s*-*t* queries. The column *dist.* refers to the time needed to recompute all distance labels from scratch.

		Preprocessing			Query Unidir.		Query Bidir.	
		time	space	dist.	# settled	time	# settled	time
graph	algorithm	[min]	[MB]	[min]	nodes	[ms]	nodes	[ms]
Europe	Dijkstra	0.0	0	0.0	9 114 385	5591.6	4 764 110	2713.2
	ALT-8	26.1	1 100	2.8	1 019 843	391.6	163 776	127.8
	ALT-16	85.2	2 200	5.5	815 639	327.6	74 669	53.6
	ALT-32	27.1	4 400	11.1	683 566	301.4	40 945	29.4
	ALT-64	68.2	8 800	22.1	604 968	288.5	25 324	19.6
USA	Dijkstra	0.0	0	0.0	11 847 523	6780.7	7 345 846	3751.4
	ALT-8	44.5	1 460	3.4	922 897	329.8	328 140	219.6
	ALT-16	103.2	2 920	6.8	762 390	308.6	180 804	129.3
	ALT-32	35.8	5 840	13.6	628 841	291.6	109 727	79.5
	ALT-64	92.9	11 680	27.2	520 710	268.8	68 861	48.9

Comparing uni- and bidirectional ALT, one may notice that the time spent per node is significantly smaller than for uni-ALT. The reason is the computational overhead for performing a bidirectional search. A reduction in search space of factor 44 (USA, ALT-16) yields a reduction in query time of factor 29. This is an overhead of factor 1.5 instead of 2.1, suggested by the figures in [15], deriving from our more efficient storage of landmark data (cf. Section 2).

Client profiles. As we do not consider repositioning landmarks, we only have to recompute all distance labels by rerunning a forward and backward Dijkstra from each landmark whenever the client profile changes. With this strategy, we are able to change a profile in 5.5 minutes when using 16 landmarks on the European network.

Updating the Preprocessing. Before testing the lazy variant of dynamic ALT, we evaluate the time needed for updating all distance labels. Note, that even the lazy variant has to update the preprocessing sometimes. With the obtained figures we want to measure the trade-off for which types of perturbations the update of the preprocessing is worth the effort. Figure 1 shows the time needed for updating all 32 trees needed for 16 landmarks if an edge is increased or decreased by factor 2 and 10. We distinguish the different types of edges.

We observe that updating the preprocessing if an important edge is altered is more expensive than the perturbation of other road types. This is due to the fact that motorway edges have many descendants within the shortest path trees. Thus, more nodes are affected by such an update. But the type of update has nearly no impact on the time spent for an update: neither how much an edge is increased nor whether an edge is increased or decreased. For almost all kind of updates we observe high fluctuations in update time. Very low update times are due to the fact that the routine is done if an increased edge is not a tree

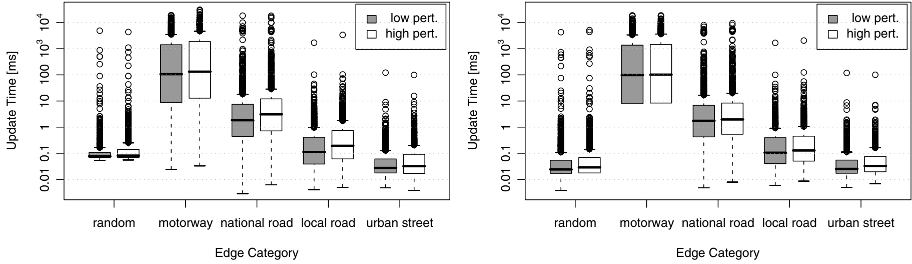


Fig. 1. Required time for updating the 32 shortest path trees needed for 16 landmarks on the European instance. The figure on the left shows the average runtime of increasing one edge by factor 2 (grey) and by 10 (white) while the right reports the corresponding values for decrementing edges. For each category, the figures are based on 10 000 edges.

edge or a decreased edge does not yield a lower distance label. Outliers of high update times are due to the fact that not only the type of the edge has an impact on the importance for updates: altering a urban street being a tree edge near a landmark may lead to a dramatic change in the structure of the tree of this landmark.

Lazy Dynamic ALT. In the following, we evaluate the robustness of the lazy variant of ALT with respect to network changes. Therefore, we alter different types and number of edges by factor 2 and factor 10.

Edge Categories. First, we concentrate on different types of edge categories. Table 2 gives an overview of the performance for both dynamic ALT variants if 1 000 edges are perturbed before running random queries.

We see that altering low-category edges has nearly no impact on the performance of lazy ALT. This is independent of the level of increase. As expected, altering motorway edges yields a loss in performance. We observe a loss of 30–45% for Europe and 15–19% for the US if the level of increase is moderate (factor 2). The situation changes for high perturbation. For Europe, queries are 3.5–5.5 times slower than in the static case (cf. Table 1), depending on the number of landmarks. The corresponding figures for the US are 1.8–2.3. Thus, lazy ALT is more robust on the US network than on the European. The loss in performance is originated from the fact that for most queries, unperturbed motorways on the shortest path have costs of 0 in the reduced graph. Thus, the search stops later if these motorways are perturbed yielding a higher search space (cf. Section 4.2). Nevertheless, comparing the query times to a bidirectional Dijkstra, we still gain a speed-up of above 10. Combining the results from Figure 1 with the ones from Table 2, we conclude that updating the preprocessing has no advantage. For motorways, updating the preprocessing is expensive and altering other types of edges has no impact on the performace of lazy ALT.

Number of Updates. In Table 2, we observed that the perturbation of motorways has the highest impact on the lazy dynamic variant of ALT. Next, we change

Table 2. Search space and query times of lazy dynamic ALT algorithm performing 10 000 random *s*-*t* queries after 1 000 edges of a specific category have been perturbed by factor 2. The figures in parentheses refer to increases by factor 10. The percentage of affected queries (the shortest path contains an updated edge) is given in column number 3.

			lazy ALT-16				lazy ALT-32			
graph	road type	aff.[%]	# settled nodes		increase [%]		# settled nodes		increase [%]	
EUR	All roads	7.5	74 700	(77 759)	0.0	(4.1)	41 044	(43 919)	0.2	(7.3)
	urban	0.8	74 796	(74 859)	0.2	(0.3)	40 996	(41 120)	0.1	(0.4)
	local	1.5	74 659	(74 669)	0.0	(0.0)	40 949	(40 995)	0.0	(0.1)
	national	28.1	74 920	(75 777)	0.3	(1.5)	41 251	(42 279)	0.7	(3.3)
	motorway	95.3	97 249	(265 472)	30.2	(255.5)	59 550	(224 268)	45.4	(447.7)
USA	All roads	3.3	181 335	(181 768)	0.3	(0.5)	110 161	(110 254)	0.4	(0.5)
	urban	0.1	180 900	(180 776)	0.1	(0.0)	109 695	(110 108)	0.0	(0.3)
	local	2.6	180 962	(181 068)	0.1	(0.1)	109 873	(109 902)	0.1	(0.2)
	national	25.5	181 490	(184 375)	0.4	(2.0)	110 553	(112 881)	0.8	(2.9)
	motorway	94.3	207 908	(332 009)	15.0	(83.6)	130 466	(247 454)	18.9	(125.5)

the number of perturbed motorways. Table 3 reports the performance of lazy dynamic ALT when different numbers of motorways are increased by factor 2 and factor 10, respectively, before running random queries on Europe.

For perturbations by factor 2, we observe almost no loss in performance for less than 500 updates, although up to 87% of the queries are affected by the perturbation. Nevertheless, 2 000 or more perturbed edges lead to significant decreases in performance, resulting in query times of about 0.5 seconds for 10 000 updates. Note that the European network contains only about 175 000 motorway edges. As expected, the loss in performance is higher when motorway edges are increased by factor 10. For this case, up to 500 perturbations can be compensated well. Comparing slight and high increases we observe that the lazy variant can

Table 3. Search space and query times of the dynamic ALT algorithm performing 10 000 random *s*-*t* queries after a variable number of motorway edges have been increased by factor 2. The figures in parentheses refer to increases by factor 10. The percentage of affected queries (the shortest path contains an updated edge) is given in column 2.

		lazy ALT-16				lazy ALT-32			
#edges	aff.[%]	# settled nodes		increase [%]		# settled nodes		increase [%]	
100	39.9	75 691	(91 610)	1.4	(22.7)	41 725	(56 349)	1.9	(37.6)
200	64.7	78 533	(107 084)	5.2	(43.4)	44 220	(69 906)	8.0	(70.7)
500	87.1	86 284	(165 022)	15.6	(121.0)	50 007	(124 712)	22.1	(204.6)
1 000	95.3	97 249	(265 472)	30.2	(255.5)	59 550	(224 268)	45.4	(447.7)
2 000	97.8	154 112	(572 961)	106.4	(667.3)	115 111	(531 801)	181.1	(1198.8)
5 000	99.1	320 624	(1 286 317)	329.4	(1622.7)	279 758	(1 247 628)	583.3	(2947.1)
10 000	99.5	595 740	(2 048 455)	697.8	(2643.4)	553 590	(1 991 297)	1252.0	(4763.3)

compensate four times more updates, e.g. 500 increases by factor 10 yield almost the same loss as 2 000 updates by factor 2.

The number of landmarks has almost no impact on the performance if more than 5 000 edges are perturbed. This is due to the fact that for almost all motorways the landmarks do not yield good reduced costs. We conclude that the lazy variant cannot compensate such a high degree of perturbation.

Comparing Lazy and Eager Dynamic ALT. Table 2 shows that lazy ALT-32 yields an increase of 40% in search space for random queries on the European network with 1 000 low perturbed motorway edges. In order to obtain a more detailed insight for which types of queries these differences are originated from, Figure 2 reports the query times of eager and lazy ALT-32 with respect to the Dijkstra rank[1] (of the target node) in this scenario.

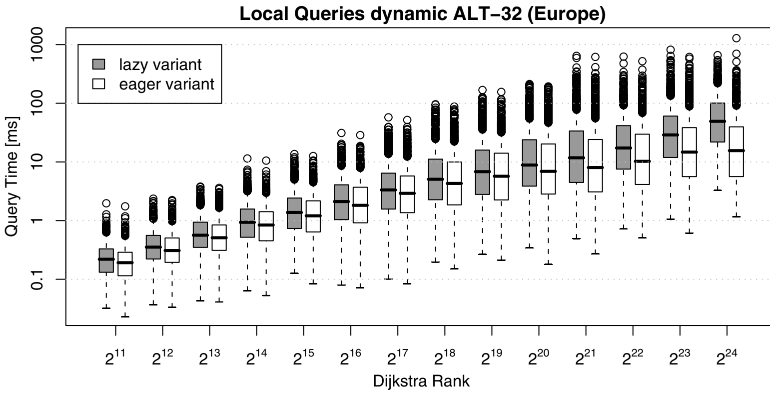


Fig. 2. Comparison of query times of the lazy and eager dynamic variant of ALT using the Dijkstra rank methodology. The queries were run after 1 000 motorways were increased by factor 2. The results are represented as box-and-whisker plot. Outliers are plotted individually.

Query performance varies so heavily that we use a logarithmic scale. For each rank, we observe queries performing 20 times worse than the median. This is originated from the fact that for some queries no landmark provides very good lower bounds resulting in significantly higher search spaces. Comparing the eager and lazy dynamic version, we observe that query times differ only by a small factor for Dijkstra ranks below 2^{20}. However, for 2^{24}, the eager version is about factor 5 faster than the lazy one. This is due to the fact that those types of queries contain a lot of motorways and most of the jammed edges are used. The eager version yields a good potential for these edge while the lazy does

[1] For an s-t query, the Dijkstra rank of node v is the number of nodes inserted in the priority queue before v is reached. Thus, it is a kind of distance measure.

not. We conclude that lazy ALT is more robust for small range queries than for long distance requests. Note that in a real-world scenario, you probably do not want to use traffic information that is more than one hour away from your current position. As the ALT algorithm provides lower bounds to *all* positions within the network, it is possible to ignore traffic jams sufficiently without any additional information.

Time-Dependent ALT. Our final experiments cover the time-dependent scenario, in which bidirectional search is prohibited. Thus, we compare time-ALT with a time-dependent variant of DIJKSTRA's algorithm [6]. This variant of DIJKSTRA works like the normal one but calculates the departure time from a node in order to use the correct edge weight. Our current implementation of time-dependency assigns 24 different transit times to each edge, representing the travel time at each hour of a day. Again, we interpret the initial values as empty roads and add transit times according to rush hours. Table 4 gives an overview of the performance on the European network for different scenarios of traffic during the day. We study three models differing in how much transit time is added to *all* edges during the rush hours. The first (high traffic) increases the transit time on all roads by factor 3 during peak hours. The low traffic scenario uses increases of factor 1.5. For comparison, the no traffic scenario uses the same (initial) edge weight for all times of the day. Our models are inspired by [8].

Table 4. Search space and query times of time-dependent ALT and DIJKSTRA performing 10 000 random s-t queries for different types of expected traffic in a time-dependent scenario. As input, the European network is used.

	no traffic		low traffic		high traffic	
algorithm	# settled	time [ms]	# settled	time [ms]	# settled	time[ms]
Dijkstra	9 029 972	8 383.2	9 034 915	8 390.6	9 033 100	8 396.1
time-ALT-16	794 622	443.7	1 969 194	1543.3	3 130 688	2 928.3

We observe a speed-up of approximately factor $3 - 5$ towards DIJKSTRA's algorithm, depending on the scenario. This relatively low speed-up in contrast to a speed-up of factor 50 for time-independent bidirectional ALT-16 towards bidirectional DIJKSTRA is due to two facts. On the one hand, the unidirectional ALT algorithm performs much worse than the bidirectional variant (see Table 1). On the other hand, lower bounds are much worse than in a time-independent scenario because an edge increased by factor 2 during rush hour counts like a perturbed edge in the time-independent scenario. As lazy ALT cannot compensate a very high degree of perturbation and we apply our traffic model to all edges including motorways, these figures are not counterintuitive.

Comparing Table 1 and 4, our time-dependent variant is 30% slower than the time-independent unidirectional ALT. This is due to overhead in computing the estimated departure time from each node.

Table 5. Comparison of lazy ALT and DynHNR [10]. 'Space' indicates the additional overhead per node. We report the performance of static and dynamic random queries. For the latter the average search space is reported after 10 and 1000 edges have been increased by factor 2 and—in parentheses—factor 10. Note that the tests for DynHNR were performed on a slightly different maschine.

	preprocessing		static queries		dynamic queries	
	time	space	time	#settled	#settled nodes	
method	[min]	[B/node]	[ms]	nodes	10 updates	1000 updates
lazy ALT-16	85	128	53.6	74 441	74 971 (75 501)	97 123 (255 754)
lazy ALT-32	27	256	29.4	40 945	41 060 (43 865)	59 550 (224 268)
lazy ALT-64	62	512	19.6	25 324	26 247 (26 901)	42 930 (201 797)
DynHNR	18	32	1.2	1 414	2 385 (8 294)	204 103 (200 465)

Comparison to Dynamic Highway-Node Routing. Analyzing the figures from Table 5, it turns out that an approach based solely on landmarks cannot compete with Dynamic Highway-Node Routing [10] as long as the number of perturbed edges stay little. However, the situation changes if more than 1000 edges are updated. For factor-2 perturbations, ALT yields lower search spaces than DynHNR and for factor-10 perturbations both techniques are very close to each other. Nevertheless, with respect to space requirements, DynHNR is superior to ALT, and the preprocessing of DynHNR can be updated more efficiently than the preprocessing of ALT.

6 Discussion

We evaluated adaptions of ALT to dynamic scenarios covering predictable and unexpected changes. In a time-independent scenario, a variant not updating the preprocessing loses almost no performance as long as the number of perturbed roads stays moderate. When using 64 landmarks, random queries are done in 20 ms on the European network and in 50 ms on the US network. However, for some types of updates the preprocessing can be updated in moderate time without storing any additional data.

Analyzing the dynamic scenarios, the time-dependent model seems to be superior to the time-independent model. Especially for long range queries, updates may occur during the traversal of the shortest path. While this can be compensated by rerunning a query from the current position, one cannot take into account jams that are on the route but probably will have disappeared as soon as you reach the critical section.

Summarizing, landmark based routing yields good query times in dynamic scenarios. Furthermore, landmarks harmonize well with other techniques like reach [15], highway hierarchies [11], or even transit nodes [4]. As the dynamization of ALT comes for free, adding landmarks to other techniques in dynamic scenarios may be worth focusing on.

Acknowledgments. We would like to thank Marco Gaertler, Robert Görke, Martin Holzer, and Bastian Katz for valueable input and proof-reading.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
2. Wagner, D., Willhalm, T.: Speed-Up Techniques for Shortest-Path Computations. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 23–36. Springer, Heidelberg (2007)
3. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-Performance Multi-Level Graphs. In: 9th DIMACS Challenge on Shortest Paths (2006)
4. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Time Shortest-Path Queries in Road Networks. In: Algorithm Engineering and Experiments (ALENEX). pp. 46–59 (2007)
5. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. pp. 156–165 (2005)
6. Cooke, K., Halsey, E.: The shortest route through a network with time-dependent intemodal transit times. Journal of Mathematical Analysis and Applications 14, 493–498 (1966)
7. Ikeda, T., Hsu, M., Imai, H., Nishimura, S., Shimoura, H., Hashimoto, T., Tenmoku, K., Mitoh, K.: A fast algorithm for finding better routes by AI search techniques. In: Vehicle Navigation and Information Systems Conference (1994)
8. Flinsenberg, I.C.M.: Route planning algorithms for car navigation. PhD thesis, Technische Universiteit Eindhoven (2004)
9. Wagner, D., Willhalm, T., Zaroliagis, C.: Geometric containers for efficient shortest-path computation. ACM Journal of Experimental Algorithmics 10, 1–30 (2005)
10. Sanders, P., Schultes, D.: Dynamic Highway-Node Routing. In: 6th Workshop on Experimental Algorithms (WEA) to appear (2007)
11. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway Hierarchies Star. In: 9th DIMACS Challenge on Shortest Paths (2006)
12. Narvaez, P., Siu, K.Y., Tzeng, H.Y.: New dynamic algorithms for shortest path tree computation. IEEE/ACM Trans. Netw. 8, 734–746 (2000)
13. Demetrescu, C., Italiano, G.F.: A new approach to dynamic all pairs shortest paths. J. ACM 51, 968–992 (2004)
14. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research (2004)
15. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Algorithm Engineering and Experimentation (ALENEX). pp. 26–40 (2005)
16. Sedgewick, R., Vitter, J.S.: Shortest paths in Euclidean space. Algorithmica 1, 31–48 (1986)
17. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In: Algorithm Engineering and Experiments (ALENEX). pp. 129–143 (2006)
18. Dashtinezhad, S., Nadeem, T., Dorohonceanu, B., Borcea, C., Kang, P., Iftode, L.: TrafficView: a driver assistant device for traffic monitoring based on car-to-car communication. In: Vehicular Technology Conference, pp. 2946–2950. IEEE, New York (2004)
19. Kaufman, D.E., Smith, R.L.: Fastest paths in time-dependent networks for intelligent-vehicle-highway systems application. IVHS Journal 1, 1–11 (1993)

Dynamic Highway-Node Routing*

Dominik Schultes and Peter Sanders

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{schultes,sanders}@ira.uka.de

Abstract. We introduce a *dynamic* technique for fast route planning in large road networks. For the first time, it is possible to handle the practically relevant scenarios that arise in present-day navigation systems: When an edge weight changes (e.g., due to a traffic jam), we can update the preprocessed information in 2–40 ms allowing subsequent fast queries in about one millisecond on average. When we want to perform only a single query, we can skip the comparatively expensive update step and directly perform a prudent query that automatically takes the changed situation into account. If the overall cost function changes (e.g., due to a different vehicle type), recomputing the preprocessed information takes typically less than two minutes.

The foundation of our dynamic method is a new static approach that generalises and combines several previous speedup techniques. It has outstandingly low memory requirements of only a few bytes per node.

1 Introduction

Computing fastest routes in road networks is one of the showpieces of real-world applications of algorithmics. In principle we could use Dijkstra's algorithm. But for large road networks this would be far too slow. Therefore, in recent years, there has been considerable interest in speed-up techniques for route planning. For an overview, we refer to [1]. The most successful methods are *static*, i.e., they assume that the network—including its edge weights—does not change. This makes it possible to *preprocess* some information *once and for all* that can be used to accelerate *all* subsequent point-to-point *queries*. Today, the static routing problem in road networks can be regarded as largely solved.

However, real road networks change all the time. In this paper, we address two such *dynamic* scenarios: individual edge weight updates, e.g., due to traffic jams, and switching between different cost functions that take vehicle type, road restrictions, or driver preferences into account.

1.1 Related Work

Bidirectional Search. simultaneously searches forward from s and backwards from t until the search frontiers meet. Many speedup techniques (including ours) use bidirectional search as an ingredient.

* Partially supported by DFG grant SA 933/1-3.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 66–79, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Separators. Perhaps the most well known property of road networks is that they are almost planar, i.e., techniques developed for planar graphs will often also work for road networks.

Using $O(n \log^2 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [2,3] for directed planar graphs without negative cycles; edge weights can be updated in amortised $O(n^{2/3} \log^{5/3} n)$ time per operation (provided that all edge weights are positive).

A previous practical approach is the *separator-based multi-level method* [4,5,6]. Out of several existing variants, we mainly refer to [5, basic variant]. For a graph $G = (V, E)$ and a node set $V' \subseteq V$, a *shortest-path overlay graph* $G' = (V', E')$ has the property that E' is a minimal set of edges such that each shortest-path distance $d(u, v)$ in G' is equal to the shortest-path distance from u to v in G. In the separator based approach, V' is chosen in such a way that the subgraph induced by $V \setminus V'$ consists of small components of similar size. The overlay graph can be constructed by performing a search in G from each separator node that stops when all neighbouring separator nodes have been found. In a bidirectional query algorithm, the components that contain the source and target nodes are searched considering *all* edges. From the border of these components, i.e., from the separator nodes, however, the search is continued considering only edges of the overlay graph. By recursing on G', this basic idea is generalised to multiple levels.

Bauer [7] observes that if the weight of an edge within some component C changes, we do not have to repeat the complete construction process of G'. It is sufficient to rerun the construction step only from some separator nodes at the boundary of C. No experimental evaluation is given. In a theoretical study on the dynamisation of shortest-path overlay graphs [8], an algorithm is presented that requires $O(|V'|(n+m)\log n)$ preprocessing time and $O(|V'|(n+m))$ space, which seems impractical for large graphs.

Highway Hierarchies. Commercial systems use information on road categories to speed up search. 'Sufficiently far away' from source and target, only 'important' roads are used. This requires manual tuning of the data and a delicate tradeoff between computation speed and suboptimality of the computed routes. In previous papers [9,10] we introduced the idea to *automatically* compute *highway hierarchies* that yield *optimal* routes *uncompromisingly quickly*. The basic idea is to define a neighbourhood for each node to consist of its H closest neighbours. Now an edge (u, v) is a highway edge if there is some shortest path $\langle s, \dots, u, v, \dots t\rangle$ such that neither u is in the neighbourhood of t nor v is in the neighbourhood of s. This defines the first level of the highway hierarchy. After contracting the network to remove low degree nodes—obtaining its so-called *core*—, the same procedure (identifying the highway network at the next level followed by contraction) is applied recursively. We obtain a hierarchy. The query algorithm is bidirectional Dijkstra with restrictions on relaxing certain edges. Roughly, far away from source or target, only high level edges need to be considered.

Reach-Based Routing [11,12] is a speed-up technique that is similar to highway hierarchies. A dynamic version that handles a set of edge weight changes is presented in [7]. The basic idea is to rerun the construction step only from nodes within a certain area, which has to be identified first. So far, the concept of *shortcuts* [9,12], which is important to get competitive construction and query times, has not been integrated in the dynamic version. No experimental evaluation for the dynamic scenario is given in [7].

Transit Node Routing. [13] is based on the following observation: "When you drive to somewhere 'far away', you will leave your current location via one of only a few 'important' traffic junctions [*transit nodes*]". Distances from each node to all neighbouring transit nodes and between all transit nodes are precomputed so that a non-local shortest-path query can be reduced to a small number of table lookups. Currently, transit node routing provides the best query times on road networks, but so far no dynamic version is available.

Goal Direction. Another interesting property of road networks is that they allow effective goal directed search using **A*** *search* [14]: lower bounds define a vertex potential that directs search towards the target. This approach was shown to be very effective if lower bounds are computed using precomputed shortest-path distances to a carefully selected set of about 20 **L***andmark* nodes [15] using the **T**riangle inequality (ALT). In [15] it is also briefly mentioned that in case of an edge weight *increase*, the query algorithm stays correct even if the landmarks and the landmark distances are not updated. To cope with drastic changes or edge weight decreases, an update of the landmark distances is suggested. In [16], these ideas are pursued leading to an extensive experimental study of landmark-based routing in various dynamic scenarios. For a comparison to our approach, see Section 5.

1.2 Overview and Main Contributions

Our first main contribution is the generalisation of the separator-based multi-level method to *highway-node routing* where overlay graphs are defined using arbitrary node sets $V' \subseteq V$ rather than separators. This requires new algorithms for preprocessing and queries since removing V' will in general *not* partition the graph into small components. Section 2 lies the ground for these new algorithms by systematically investigating the graph theoretical problem of finding all nodes from V' that can be reached on a shortest path from a given node without passing another node from V'. In Section 3 we apply the results to static highway-node routing. The main remaining difficulty is to choose the highway nodes. The idea is that *important* nodes used by many shortest paths will lead to overlay graphs that are more sparse than for the separator-based approach. This will result in faster queries and low space consumption. The intuition behind this idea is that the number of overlay graph edges needed between the separator nodes bordering a region grows quadratically with the number of border nodes (see also [6]). In contrast, important nodes are uniformly distributed over the

network and connected to only a small number of nearby important nodes (see also [13]). Indeed, our method is so far the most space-efficient preprocessing technique that allows query times several thousand times faster than Dijkstra's algorithm. A direct comparison to the separator-based variant is difficult since previous papers use comparatively small graphs[1] and it is not clear how the original approach scales to very large graphs.

While there are many ways to choose important nodes, we capitalise on previous results and use *highway hierarchies* to define all required node sets. There is an analogy to *transit node routing* where we also used highway hierarchies to find important nodes.

On the first glance, our approach to highway-node routing looks like a roundabout way to achieve similar results as with the direct application of highway hierarchies. However, by factoring out all the complications of highway hierarchies into a pre-preprocessing step, we simplify data structures, the query algorithm, and, most importantly, *dynamic variants*. The idea is that in practice, a set of nodes important for one weight function will also contain most of the important nodes for another 'reasonable' weight function. The advantage is obvious when the cost function is redefined—all we have to do is to recompute the edge sets of the overlay graphs. Section 4 discusses two variants of the scenario when a few edge weights change: In a server setting, the overlay graphs are updated so that afterwards the static query algorithm will again yield correct results. In a mobile setting, the data structure are not updated. Rather, the query algorithm searches at lower levels of the node hierarchy, (only) where the information at the higher levels might have been compromised by the changed edges.

Together with [16], we are the first to present an approach that tackles such dynamic scenarios and to demonstrate its efficiency in an extensive experimental study using a real-world road network, which is summarised in Section 5.

2 Covering Nodes

Dijkstra's Algorithm can be used to solve the *single-source shortest-path problem*, i.e., to compute the shortest paths from a single source node s to all other nodes in a given graph. Starting with the source node s as root, Dijkstra's algorithm grows a *shortest-path tree* T that contains shortest paths from s to all other nodes. During this process, each node of the graph is either *unreached*, *reached*, or *settled*. A node that already belongs to the tree is *settled*. If a node u is settled, a shortest path P^* from s to u has been found and the distance $d(s, u) = w(P^*)$ is known. A node that is adjacent to a settled node is *reached*. Note that a settled node is also reached. If a node u is reached, a path P from s to u, which might not be the shortest one, has been found and a *tentative*

[1] For a subgraph of the European road network with about 100 000 nodes, [6] gives a preprocessing time of "well over half an hour [plus] several minutes" and a query time 22 times faster than Dijkstra's algorithm. For a comparison, we take a subgraph around Karlsruhe of a very similar size, which we preprocess in seven seconds. Then, we obtain a speedup of 94.

distance $\delta(u) = w(P)$ is known. Reached but not settled nodes are also called *queued*. Nodes that are not reached are *unreached*.

Problem Definition. During a Dijkstra search from s, we say that a settled node u is *covered* by a node set V' if there is at least one node $v \in V'$ on the path from the root s to u. A queued node is *covered* if its tentative parent is covered. The current partial shortest-path tree T is *covered* if all currently queued nodes are covered. All nodes $v \in V' \cap T$ that have no parent in T that is covered are *covering nodes*, forming the set $\mathcal{C}_G(V', s)$.

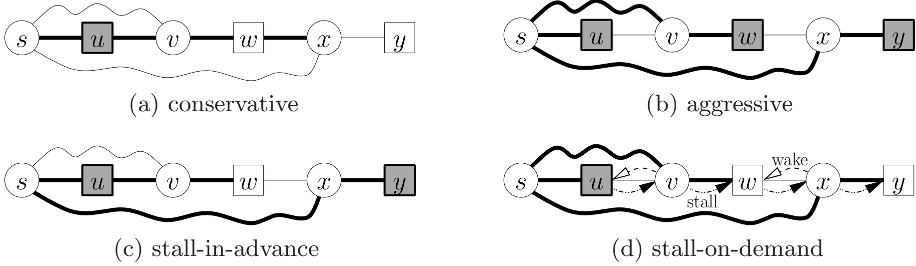
The crucial subroutine of all algorithms in the subsequent sections takes a graph G, a node set V', and a root s and determines all covering nodes $\mathcal{C}_G(V', s)$. We distinguish between four different ways of doing this.

Conservative Approach. The *conservative* variant (Fig. 1(a)) works in the obvious way: a search from s is stopped as soon as the current partial shortest-path tree T is covered. Then, it is straightforward to read off all covering nodes. However, if the partial shortest-path tree contains one path that is not covered for a long time, the tree can get very big even though all other branches might have been covered very early. In our application, this is a critical issue due to long-distance ferry connections.

Aggressive Approach. As an overreaction to the above observation, we might want to define an *aggressive* variant that does not continue the search from any covering node, i.e., some branches might be terminated early, while only the non-covered paths are followed further on. Unfortunately, this provokes two problems. First, we can no longer guarantee that T contains only shortest paths. As a consequence, we get a superset $\overline{\mathcal{C}}_G(V', s)$ of the covering nodes, which still can be used to obtain correct results. However, the performance will be impaired. In Section 3, we will explain how to reduce a given superset rather efficiently in order to obtain the exact covering node set. Second, the tree T can get even bigger since the search might continue *around* the covering nodes where we pruned the search.[2] In our example (Fig. 1(b)), the search is pruned at u so that v is reached using a much longer path that leads around u. As a consequence, w is superfluously marked as a covering node.

Stall-in-Advance Technique. If we decide not to prune the search immediately, but to go on 'for a while' in order to *stall* other branches, we obtain a compromise between the conservative and the aggressive variant, which we call *stall-in-advance*. One heuristic we use prunes the search at node z when the path explored from s to z contains p nodes of V' for some tuning parameter p. Note that for $p := 1$, the stall-in-advance variant corresponds to the aggressive variant. In our example (Fig. 1(c)), we use $p := 2$. Therefore, the search is pruned

[2] Note that the query algorithm of the separator-based approach virtually uses the aggressive variant to compute covering nodes. This is reasonable since the search can never 'escape' the component where it started.



(a) conservative (b) aggressive

(c) stall-in-advance (d) stall-on-demand

Fig. 1. Simple example for the computation of covering nodes. We assume that all edges have weight 1 except for the edges (s, v) and (s, x), which have weight 10. In each case, the search process is started from s. The set V' consists of all nodes that are represented by a square. Thick edges belong to the search tree T. Nodes that belong to the computed superset $\overline{\mathcal{C}}_G(V', s)$ of the covering nodes are highlighted in grey. Note that the actual covering node set contains only one node, namely u.

not until w is settled. This stalls the edge (s, v) and, in contrast to (b), the node v is covered. Still, the search is pruned too early so that the edge (s, x) is used to settle x.

Stall-on-Demand Technique. In the stall-in-advance variant, relaxing an edge leaving a covered node is based on the 'hope' that this might stall another branch. However, our heuristic is not perfect, i.e., some edges are relaxed in vain, while other edges which would have been able to stall other branches, are not relaxed. Since we are not able to make the perfect decision in advance, we introduce a fourth variant, namely *stall-on-demand*. It is an extension of the aggressive variant, i.e., at first, edges leaving a covered node are not relaxed. However, if such a node u is reached later via another path, it is *woken up* and a breadth-first search (BFS) is performed from that node: an adjacent node v that has already been reached by the main search is inserted into the BFS queue if we can prove that the best path P found so far is suboptimal. This is certainly the case if the path from s via u to v is shorter than P. All nodes encountered during the BFS are marked as *stalled*. The main search is pruned at stalled nodes. Furthermore, stalled nodes are never marked as covering nodes. The stalling process cannot invalidate the correctness since only nodes are stalled that otherwise would contribute to suboptimal paths. In our example (Fig. 1(d)), the search is pruned at u. When v is settled, we assume that the edge (v, w) is relaxed first. Then, the edge (v, u) wakes the node u up. A stalling process (a BFS search) is started from u. The nodes v and w are marked as stalled. When w is settled, its outgoing edges are not relaxed. Similarly, the edge (x, w) wakes the stalled node w and another stalling process is performed.

3 Static Highway-Node Routing

Multi-Level Overlay Graph. For given *highway-node sets* $V =: V_0 \supseteq V_1 \supseteq ... \supseteq V_L$, we give a definition of the *multi-level overlay graph* $\mathcal{G} = (G_0, G_1, ..., G_L)$

that is almost equivalent to the definition in [5]: $G_0 := G$ and for each $\ell > 0$, we have $G_\ell := (V_\ell, E_\ell)$ with $E_\ell := \{(s,t) \in V_\ell \times V_\ell \mid \exists \text{ shortest path } P = \langle s, u_1, u_2, \ldots, u_k, t\rangle \text{ in } G_{\ell-1} \text{ s.t. } \forall i : u_i \notin V_\ell\}$.

Node Selection. We can choose any highway-node sets to get a correct procedure. However, this choice has a big impact on preprocessing and query performance. Roughly speaking, a node that lies on many shortest paths should belong to the node set of a high level. In a first implementation, we use the set of level-ℓ core nodes of the highway hierarchy of G as highway-node set V_ℓ. In other words, we let the construction procedure of the highway hierarchies decide the importance of the nodes.

Construction. The multi-level overlay graph is constructed in a bottom-up fashion. In order to construct level $\ell > 0$, we determine for each node $s \in V_\ell$ its covering node set $\mathcal{C} := \mathcal{C}_{G_{\ell-1}}(V_\ell \setminus \{s\}, s)$. For each $u \in \mathcal{C}$, we add an edge with weight $d(s,u)$ to E_ℓ. The stall-in-advance and the stall-on-demand variant introduced in Section 2 are efficient algorithms for computing a *superset* of $\mathcal{C}$, implying that they add some superfluous edges. Optionally, we can apply the following *reduction step* to eliminate those edges: for each node $u \in V_\ell$, we perform a search in G_ℓ (instead of $G_{\ell-1}$) until all adjacent nodes have been settled. For any node v that has been settled via a path that consists of more than one edge, we can remove the edge (u,v) since a (better) alternative that does not require this edge has been found.

Query. The query algorithm is a symmetric bidirectional procedure so that it suffices to describe the forward search, which works in a bottom-up fashion. For conciseness, we only describe the variant based on stall-on-demand. We perform a modified Dijkstra search from s in $(V, E_0 \cup \cdots \cup E_L)$. From a node whose highest level is i, only edges in $E_i \cup \cdots \cup E_L$ are relaxed. When an edge in $E_0 \cup \cdots \cup E_{i-1}$ reaches a node v in V_i, v is 'woken up' and performs a BFS in the shortest path tree to stall nodes that are not on shortest paths.

Forward and backward search are interleaved. We keep track of a tentative shortest-path length resulting from nodes that have been settled in both search directions. We abort the forward (backward) search when all keys in the forward (backward) priority queue are greater than the tentative shortest-path length.

4 Dynamic Highway-Node Routing

When a small set of edges change their weight, we can distinguish between a *server scenario* and a *mobile scenario*: In the former, a server has to react to incoming events by updating its data structures so that *any* point-to-point query can be answered correctly; in the latter, a mobile device has to react to incoming events by (re)computing a *single* point-to-point query taking into account the new situation. In the server scenario, it pays to invest some time to perform the update operation since a lot of queries depend on it. In the mobile scenario, we

do not want to waste time for updating parts of the graph that are irrelevant to the current query.

Server Scenario. Similar to an exchange of the cost function, when a single or a few edge weights change, we keep the highway-node sets and update only the overlay graphs. In this case, however, we do not have to repeat the complete construction from scratch, but it is sufficient to perform the construction step only from nodes that might be affected by the change. Certainly, a node v whose partial shortest-path tree of the initial construction did not contain any node u of a modified edge (u, x) is *not* affected: if we repeated the construction step from v, we would get exactly the same partial shortest-path tree and, consequently, the same result.

During the first construction (and all subsequent update operations), we manage sets A_u^ℓ of nodes whose level-ℓ preprocessing might be affected when an outgoing edge of u changes: when a level-ℓ construction step from some node v is performed, for each node u in the partial shortest-path tree[3], we add v to A_u^ℓ. Note that these sets can be stored explicitly (as we do it in our current implementation) or we could store a superset, e.g., by some kind of *geometric container* (a disk, for instance). Figure 2 contains the pseudo-code of the update algorithm.

```
input: set of edges E^m with modified weight;
define set of modified nodes: V_0^m := {u | (u,v) ∈ E^m};
foreach level ℓ, 1 ≤ ℓ ≤ L, do
    V_ℓ^m := ∅; R_ℓ := ⋃_{u ∈ V_{ℓ-1}^m} A_u^ℓ;
    foreach node v ∈ R_ℓ do
        repeat construction step from v;
        if something changes, put v to V_ℓ^m;
```

Fig. 2. The update algorithm that deals with a set of edge weight changes

Mobile Scenario. In the mobile scenario, we only determine the sets of potentially unreliable nodes by using a fast variant of the update algorithm (Fig. 2), where from the last two lines only the "put v to V_ℓ^m" is kept. (Note that in particular the construction step is *not* repeated.) Then, for each node $u \in V$, we define the *reliable level* $r(u) := \min\{i - 1 \mid u \in R_i\}$ with $\min \emptyset := \infty$. In order to get correct results without updating the data structures, the query algorithm has to be modified. First, we do not relax any edge (u, v) that has been created

[3] When the stall-in-advance technique is used, some nodes are only added to the tree to potentially stall other branches. Upon completion of the construction step, we can identify nodes that have been added in vain, i.e., that were not able to stall other branches. Those nodes had no actual influence on the construction and, thus, can be ignored at this point.

during the construction of some level $> r(u)$. Second, if the search at some node u has already reached a level $\ell > r(u)$, then the search at this node is *downgraded* to level $r(u)$. In other words, if we arrive at some node from which we would have to repeat the construction step, we do not use potentially unreliable edges, but continue the search in a sufficiently low level to ensure that the correct path can be found.

Note that the update procedure, which is also used to determine the sets of potentially unreliable nodes, is performed in the forward direction. Its results cannot be directly applied to the backward direction of the query. It is simple to adjust the first modification to this situation (by considering $r(v)$ instead of $r(u)$ when dealing with an edge (u, v)). Adjusting the second modification would be more difficult, but fortunately we can completely omit it for the backward direction. As a consequence, the search process becomes asymmetric. While the forward search is continued in lower levels whenever it is necessary, the backward search is never downgraded. If 'in doubt', the backward search stops and waits for the forward search to finish the job.

5 Experiments

Environment and Instances. The experiments were done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GB main memory and 2×1 MB L2 cache, running SuSE Linux 10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3.

We deal with the road network of Western Europe[4], which has been made available for scientific use by the company PTV AG. It consists of 18 029 721 nodes and 42 199 587 directed edges. The original graph contains for each edge a length and a road category. There are four major road categories (motorway, national road, regional road, urban street), which are divided into three subcategories each. In addition, there is one category for forest and gravel roads. We assign average speeds (130, 120, . . ., 10 km/h) to the road categories, compute for each edge the average travel time, and use it as weight. We call this our *default* speed profile. Experiments which we did on a US and Canadian road network of roughly the same size (provided by PTV as well) show exactly the same relative behaviour as in [10], namely that it is slightly more difficult to handle North America than Europe (e.g., 20% slower query times). Due to space constraints, we give detailed results only for Europe.

For now, we report the times needed to compute the shortest-path distance between two nodes without outputting the actual route. Note that we could also output full path descriptions using the techniques from [17], expecting a similar performance as in [17]. The query times are averages based on 10 000 randomly chosen (s, t)-pairs. In addition to providing average values, we use the methodology from [9] in order to plot query times against the 'distance' of the target from the source, where in this context, the *Dijkstra rank* is used as a measure of distance: for a fixed source s, the Dijkstra rank of a node t

[4] 14 countries: at, be, ch, de, dk, es, fr, it, lu, nl, no, pt, se, uk.

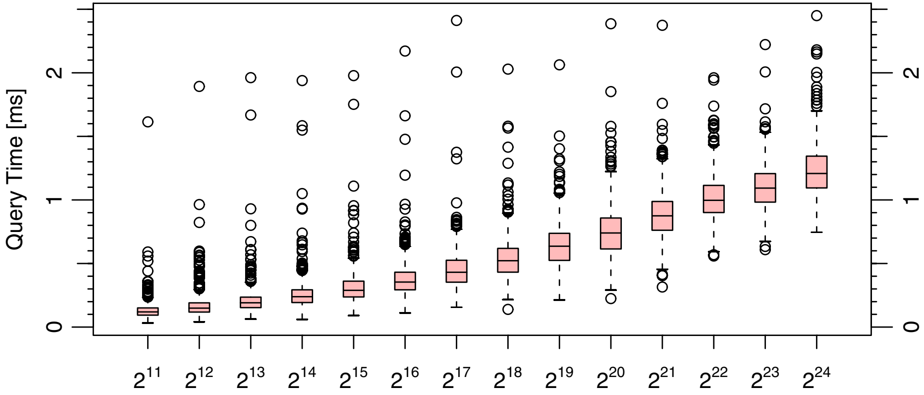


Fig. 3. Query performance against Dijkstra rank for the default speed profile, with edge reduction. Each box represents the three quartiles [18, box-and-whisker plot].

is the rank w.r.t. the order which Dijkstra's algorithm settles the nodes in. Such plots are based on 1 000 random source nodes. After performing a lot of preliminary experiments, we decided to apply the stall-in-advance technique to the construction and update process (with $p := 1$ for the construction of level 1 and $p := 5$ for all other levels) and the stall-on-demand technique to the query.

Highway Hierarchy Construction. In order to determine the highway node sets, we construct seven levels of the highway hierarchy using our default speed profile and neighbourhood size[5] $H = 70$. This can be done in 16 minutes. For *all* further experiments, these highway-node sets are used.

Static Scenario. The first data column of Tab. 1 contains the construction time of the multi-level overlay graph and the average query performance for the default speed profile. Figure 3 shows the query performance against the Dijkstra rank. The disk space overhead of the static variant is 8 bytes per node to store the additional edges of the multi-level overlay graph and the level data associated with the nodes. Note that this overhead can be further reduced to as little as 2.0 bytes per node yielding query times of 1.55 ms (Tab. 4). The total disk space[6] of 32 bytes per node also includes the original edges and a mapping from original to internal node IDs (that is needed since the nodes are reordered by level).

Changing the Cost Function. In addition to our default speed profile, Tab. 1 also gives the construction and query times for a few other selected speed profiles (which have been provided by the company PTV AG) using the same highway-node sets. Note that for most road categories, our profile is slightly faster than

[5] For details on the highway hierarchy construction and the definition of *neighbourhood size*, we refer to [9,10].

[6] The main memory usage is somewhat higher. However, we cannot give exact numbers for the static variant since our implementation does not allow to switch off the dynamic data structures.

Table 1. Construction time of the overlay graphs and query performance for different speed profiles using the same highway-node sets. For the default speed profile, we also give results for the case that the edge reduction step (Section 3) is applied.

speed profile	default	(reduced)	fast car	slow car	slow truck	distance
constr. [min]	1:40	(3:04)	1:41	1:39	1:36	3:56
query [ms]	1.17	(1.12)	1.20	1.28	1.50	35.62
#settled nodes	1 414	(1 382)	1 444	1 507	1 667	7 057

Table 2. Update times per changed edge [ms] for different road types and different update types: add a traffic jam (+), cancel a traffic jam (−), block a road (∞), and multiply the weight by 10 (×). Due to space constraints, some columns are omitted.

	any road type				motorway				national		regional		urban	
|change set|	+	−	∞	×	+	−	∞	×	+	∞	+	∞	+	∞
1	2.7	2.5	2.8	2.6	40.0	40.0	40.1	37.3	19.9	20.3	8.4	8.6	2.1	2.1
1000	2.4	2.3	2.4	2.4	8.4	8.1	8.3	8.1	7.1	7.1	5.3	5.3	2.0	2.0

Table 3. Query performance depending on the number of edge weight changes (select only motorways, multiply weight by 10). For ≤ 100 changes, 100 different edge sets are considered; for ≥ 1 000 changes, we deal only with one set. For each set, 1 000 queries are performed. We give the average percentage of queries whose shortest-path length is affected by the changes, the average number of settled nodes (also relative to zero changes), and the average query time, broken down into the init phase where the reliable levels are determined and the search phase.

	affected	#settled nodes		query time [ms]		
|change set|	queries	absolute	relative	init	search	total
1	0.6 %	2 347	(1.7)	0.3	2.0	2.3
10	6.3 %	8 294	(5.9)	1.9	7.2	9.1
100	41.3 %	43 042	(30.4)	10.6	36.9	47.5
1 000	82.6 %	200 465	(141.8)	62.0	181.9	243.9
10 000	97.5 %	645 579	(456.6)	309.9	627.1	937.0

PTV's fast car profile. The last speed profile ('distance') virtually corresponds to a distance metric since for each road type the same constant speed is assumed. The performance in case of the three PTV travel time profiles is quite close to the performance for the default profile. Hence, we can switch between these profiles without recomputing the highway-node sets. The constant speed profile is a rather difficult case. Still, it would not completely fail, although the performance gets considerably worse. We assume that any other 'reasonable' cost function would rank somewhere between our default and the constant profile.

Updating a Few Edge Weights (Server Scenario). In the dynamic scenario, we need additional space to manage the affected node sets A_u^ℓ. Furthermore, the edge reduction step is not yet supported in the dynamic case so that the total disk space usage increases to 56 bytes per node. In contrast to the static

Table 4. Comparison between pure highway hierarchies [17], three variants of highway-node routing (HNR), and dynamic ALT-16 [16]. 'Space' denotes the average disk space *overhead*. We give execution times for both a complete recomputation using a similar cost function and an update of a single motorway edge multiplying its weight by 10. Furthermore, we give search space sizes after 10 and 1 000 edge weight changes (motorway, ×10) for the mobile scenario. Time measurements in parentheses have been obtained on a similar, but not identical machine.

	preprocessing		static queries		updates		dynamic queries	
	time	space	time	#settled	compl.	single	#settled nodes	
method	[min]	[B/node]	[ms]	nodes	[min]	[ms]	10 chgs.	1000 chgs.
HH pure	17	28	1.16	1 662	17	–	–	–
StHNR	19	8	1.12	1 382	3	–	–	–
StHNR mem	24	2	1.55	2 453	8	–	–	–
DynHNR	18	32	1.17	1 414	2	37	8 294	200 465
DynALT-16	(85)	128	(53.6)	74 441	(6)	(2 036)	75 501	255 754

variant, the main memory usage is considerably higher than the disk space usage (around a factor of two) mainly because the dynamic data structures maintain vacancies that might be filled during future update operations.

We can expect different performances when updating very important roads (like motorways) or very unimportant ones (like urban streets, which are usually only relevant to very few connections). Therefore, for each of the four major road categories, we pick 1 000 edges at random. In addition, we randomly pick 1 000 edges irrespective of the road type. For each of these edge sets, we consider four types of updates: first, we add a traffic jam to each edge (by increasing the weight by 30 minutes); second, we cancel all traffic jams (by setting the original weights); third, we block all edges (by increasing the weights by 100 hours, which virtually corresponds to 'infinity' in our scenario); fourth, we multiply the weights by 10 in order to allow comparisons to [16]. For each of these cases, Tab. 2 gives the average update time per changed edge. We distinguish between two change set sizes: dealing with only one change at a time and processing 1 000 changes simultaneously.

As expected, the performance depends mainly on the selected edge and hardly on the type of update. The average execution times for a single update operation range between 40 ms (for motorways) and 2 ms (for urban streets). Usually, an update of a motorway edge requires updates of most levels of the overlay graph, while the effects of an urban-street update are limited to the lowest levels. We get a better performance when several changes are processed at once: for example, 1 000 random motorway segments can be updated in about 8 seconds. Note that such an update operation will be even more efficient when the involved edges belong to the same local area (instead of being randomly spread), which might be a common case in real-world applications.

Updating a Few Edge Weights (Mobile Scenario). Table 3 shows for the most difficult case (updating motorways) that using our modified query

algorithm we can omit the comparatively expensive update operation and still get acceptable execution times, at least if only a moderate amount of edge weight changes occur. Additional experiments have confirmed that, similar to the results in Tab. 2, the performance does not depend on the update type (add 30 minutes, multiply by 10, ...), but on the edge type (motorway, urban street, ...) and, of course, on the number of updates.

Comparisons. Highway-node routing has similar preprocessing and query times as pure highway hierarchies [17, w/o distance table], but (in the static case) a significantly smaller memory overhead. Table 4 gives detailed numbers, and it also contains a comparison to the dynamic ALT approach [16] with 16 landmarks. We can conclude that as a stand-alone method, highway-node routing is (clearly) superior to dynamic ALT w.r.t. all studied aspects.[7]

6 Conclusion

Combining and considerably extending ideas of several previous static point-to-point route planning techniques yields a new static approach with extremely low space requirements, fast preprocessing, and very good query times. More important and innovative, however, is the fact that the approach can be extended to work in dynamic scenarios: we can efficiently react to events like traffic jams. Furthermore, we deal with the case that different cost functions are handled.

There is room to improve the performance both at the implementation and at the algorithmic level. In particular, better ways to select the highway-node sets might be found. The memory consumption of the dynamic variant can be considerably reduced by using a more space-efficient representation of the affected node sets. There is also room for additional functionality. For instance, we can adapt the algorithm from [19] to work with the new approach in order to support *many-to-many* shortest-path computations for exchangeable cost functions. An extension to a *time-dependent* scenario—where the edge weights depend on the time of day according to some function known in advance—is an important open problem.

References

1. Sanders, P., Schultes, D.: Engineering fast route planning algorithms. In: 6th Workshop on Experimental Algorithms (2007)
2. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci. 72(5), 868–889 (2006)
3. Klein, P.: Multiple-source shortest paths in planar graphs. In: 16th ACM-SIAM Symposium on Discrete Algorithms, SIAM, pp. 146–155 (2005)

[7] Note that our comparison concentrates on only one variant of dynamic ALT: different landmark sets can yield different tradeoffs. Also, better results can be expected when a lot of very small changes are involved. Moreover, dynamic ALT can turn out to be very useful in combination with other dynamic speedup techniques yet to come.

4. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: an empirical case study from public railroad transport. ACM Journal of Experimental Algorithmics 5, 12 (2000)
5. Holzer, M., Schulz, F., Wagner, D.: Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. In: Workshop on Algorithm Engineering and Experiments. In: Proceedings in Applied Mathematics. SIAM pp. vol. 129, pp. 156–170 (2006)
6. Holzer, M., Schulz, F., Wagner, D.: Engineering multi-level overlay graphs for shortest-path queries. invited for ACM Journal of Experimental Algorithmics (special issue Alenex 2006) (2007)
7. Bauer, R.: Dynamic speed-up techniques for Dijkstra's algorithm. Diploma Thesis, Universität Karlsruhe (TH) (2006)
8. Bruera, F., Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D.: On the dynamization of shortest path overlay graphs. Technical Report 0026, ARRIVAL (2006)
9. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 568–579. Springer, Heidelberg (2005)
10. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)
11. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments. pp. 100–111 (2004)
12. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments, Miami pp. 129–143 (2006)
13. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: Intransit to constant time shortest-path queries in road networks. In: Workshop on Algorithm Engineering and Experiments (2007)
14. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on System Science and Cybernetics 4(2), 100–107 (1968)
15. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research (2004)
16. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: 6th Workshop on Experimental Algorithms (2007)
17. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge, http://www.dis.uniroma1.it/~challenge9/ (2006)
18. R Development Core Team: R: A Language and Environment for Statistical Computing. http://www.r-project.org (2004)
19. Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D.: Computing many-to-many shortest paths using highway hierarchies. In: Workshop on Algorithm Engineering and Experiments (2007)

Dynamic Trees in Practice*

Robert E. Tarjan[1,2] and Renato F. Werneck[3]

[1] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA
ret@cs.princeton.edu
[2] Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304, USA
[3] Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA
renatow@microsoft.com

Abstract. Dynamic tree data structures maintain forests that change over time through edge insertions and deletions. Besides maintaining connectivity information in logarithmic time, they can support aggregation of information over paths, trees, or both. We perform an experimental comparison of several versions of dynamic trees: ST-trees, ET-trees, RC-trees, and two variants of top trees (self-adjusting and worst-case). We quantify their strengths and weaknesses through tests with various workloads, most stemming from practical applications. We observe that a simple, linear-time implementation is remarkably fast for graphs of small diameter, and that worst-case and randomized data structures are best when queries are very frequent. The best overall performance, however, is achieved by self-adjusting ST-trees.

1 Introduction

The *dynamic tree problem* is that of maintaining an n-vertex forest that changes over time. Edges can be added or deleted in any order, as long as no cycle is ever created. In addition, data can be associated with vertices, edges, or both. This data can be manipulated individually (one vertex or edge at a time) or in bulk, with operations that deal with an entire path or tree at a time. Typical operations include finding the minimum-cost edge on a path, adding a constant to the costs of all edges on a path, and finding the maximum-cost vertex in a tree. The dynamic tree problem has applications in network flows [14,22,25], dynamic graphs [9,11,12,15,20,28], and other combinatorial problems [16,17,18].

Several well-known data structures can (at least partially) solve the dynamic tree problem in $O(\log n)$ time per operation: ST-trees [22,23], topology trees [11,12,13], ET-trees [15,25], top trees [4,6,26,27], and RC-trees [1,2]. They all map an arbitrary tree into a balanced one, but use different techniques to achieve this goal: path decomposition (ST-trees), linearization (ET-trees), and tree contraction (topology trees, top trees, and RC-trees). As a result, their relative performance depends significantly on the workload. We consider nine variants

* Part of this work was done while the second author was at Princeton University. Research partially supported by the Aladdin Project, NSF Grant 112-0188-1234-12.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 80–93, 2007.
© Springer-Verlag Berlin Heidelberg 2007

of these data structures. Section 2 presents a high-level description of each of them, including their limitations and some implementation issues. (A more comprehensive overview can be found in [27, Chapter 2].)

Our experimental analysis, presented in Section 3, includes three applications (maximum flows, online minimum spanning forests, and a simple shortest path algorithm), as well as randomized sequences of operations. Being relatively simple, these applications have dynamic tree operations as their bottleneck. We test different combinations of operations (queries and structural updates), as well as different types of aggregation (over paths or entire trees). The experiments allow for a comprehensive assessment of the strengths and weaknesses of each strategy and a clear separation between them. Section 4 summarizes our findings and compares them with others reported in the literature.

2 Data Structures

ET-trees. The simplest (and most limited) dynamic-tree data structures are ET-trees [15,25]. They represent an *Euler tour* of an arbitrary unrooted tree, i.e., a tour that traverses each edge of the tree twice, once in each direction. It can be thought of as a circular list, with each node representing either an arc (one of the two occurrences of an edge) or a vertex of the forest. For efficiency, the list is broken at an arbitrary point and represented as a binary search tree, with the nodes appearing in symmetric (left-to-right) order. This allows edge insertions (*link*) and deletions (*cut*) to be implemented in $O(\log n)$ time as *joins* and *splits* of binary trees. Our implementation of ET-trees (denoted by ET) follows Tarjan's specification [25] and uses splay trees [23], a self-adjusting form of binary search trees whose performance guarantees are amortized.

As described by Tarjan [25], ET-trees associate values with vertices. Besides allowing queries and updates to individual values, the ET-tree interface also has operations to add a constant to all values in a tree and to find the minimum-valued vertex in a tree. These operations take $O(\log n)$ time if every node stores its value in *difference form*, i.e., relative to the value of its parent in the binary tree; this allows value changes at the root to implicitly affect all descendants. ET-trees can be adapted to support other types of queries, but they have a fundamental limitation: information can only be aggregated over trees. Efficient path-based aggregation is impossible because the two nodes representing an edge may be arbitrarily far apart in the data structure.

ST-trees. The best-known dynamic tree data structures supporting path operations are Sleator and Tarjan's ST-trees (also known as *link-cut trees*). They predate ET-trees [22,23], and were the first to support dynamic-tree operations in logarithmic time. ST-trees represent rooted trees. The basic structural operations are $link(v, w)$, which creates an arc from a root v to a vertex w, and $cut(v)$, which deletes the arc from v to its parent. The root can be changed by $evert(v)$, which reverses all arcs on the path from v to the previous root. Functions $findroot(v)$ and $parent(v)$ can be used to query the structure of the tree. As described in [23], ST-trees associate a cost with each vertex v, retrievable by

the *findcost*(v) function. Two operations deal with the path from v to the root of its tree: *addcost*(v, x) adds x to the cost of every vertex on this path, and *findmin*(v) returns its minimum-cost vertex.

The obvious implementation of the ST-tree interface is to store with each node its value and a pointer to its parent. This supports *link*, *cut*, *parent* and *findcost* in constant time, but *evert*, *findroot*, *findmin* and *addcost* must traverse the entire path to the root. Despite the linear-time worst case, this representation might be good enough for graphs with small diameter, given its simplicity. We tested two variants of this implementation, LIN-V and LIN-E; the former associates costs with vertices, the latter with edges. Both store values at vertices, but LIN-E interprets such a value as the cost of the arc to the parent (the values must be moved during *evert*).

To achieve sublinear time per operation, Sleator and Tarjan propose an indirect representation that partitions the underlying tree into vertex-disjoint *solid paths* joined by *dashed edges*. Each solid path is represented by a binary search tree where the original nodes appear in symmetric order. These binary trees are then "glued" together to create a single *virtual tree*: the root of a binary tree representing a path P becomes a *middle child* (in the virtual tree) of the parent (in the original forest) of the topmost vertex of P. For an efficient implementation of *addcost* and *evert*, values are stored in difference form.

To manipulate a path from v to the root, the data structure first *exposes* it, i.e., changes the partition of the tree (by a series of *joins* and *splits* of binary trees) so that the unique solid path containing the root starts at v. Standard binary tree operations can then be applied to the exposed path to implement *findroot*, *parent* and *findmin* in logarithmic time. When paths are represented as splay trees, *expose* and the other dynamic tree operations run in $O(\log n)$ amortized time [23]. A worst-case logarithmic bound is achievable with globally biased search trees [22], but this solution is too complicated to be practical.

We call the splay-based implementation used in our experiments ST-V. Although it has costs on vertices, it can also represent costs on edges as long as *evert* is never called. Supporting *evert* with costs on edges requires maintaining additional nodes to represent the edges explicitly. Our implementation of this variant, ST-E, uses ST-V as the underlying data structure.

ST-trees can be modified to support other types of queries, as long as information is aggregated over paths only. Aggregation over arbitrary trees would require traversing the ST-tree in a top-down fashion, which is impossible because nodes do not maintain pointers to their (potentially numerous) middle children. A solution is to apply *ternarization* to the underlying forest, which replaces each high-degree vertex by a chain of low-degree ones [14,16,17,18,20].

Tree contraction. A third approach is to represent a *contraction* of the tree, as done by *topology trees*, *RC-trees*, and *top trees*. We concentrate on the most general, top trees, and briefly discuss the other two.

Top trees were introduced by Alstrup et al. [4,6], but we borrow the notation used by Tarjan and Werneck [26]. The data structure represents free (unrooted) trees with sorted adjacency lists (i.e., the edges adjacent to each vertex are

arranged in some fixed circular order, which can be arbitrary). A *cluster* represents both a subtree and a path of the original tree. Each original edge of the graph is a *base cluster*. A *tree contraction* is a sequence of local operations that successively pair up these clusters until a single cluster remains. The *top tree* is merely the binary tree representing the contraction. If two clusters (u, v) and (v, w) share a degree-two endpoint v, they can be combined into a *compress cluster* (u, w). Also, if (w, x) is the successor of (v, x) (in the circular order around x) and v has degree one, these clusters can be combined into a *rake cluster*, also with endpoints w and x. Each *rake* or *compress* cluster can be viewed as a parent that aggregates the information contained in its children. It represents both the subtree induced by its descendants and the path between its two endpoints, and can also be viewed as a higher-level edge. The root of the top tree represents the entire underlying tree. Whenever there is a *link* or *cut*, the data structure merely updates the contractions affected.

An important feature of the top tree interface is that it decouples the contraction itself from the values that must be manipulated. The data structure decides which operations (*rake* or *compress*) must be performed, but updates no values on its own. Instead, it calls user-defined *internal functions* to handle value updates whenever a pairing of clusters is performed (*join*) or undone (*split*). The interface stipulates that these calls will be made when all clusters involved are *roots* of (maybe partial) top trees—none of the input clusters will have a parent. This makes the implementation of the call-back functions easier, but it may hurt performance: because *joins* must be called bottom-up and *splits* top-down, the tree cannot be updated in a single pass.

To perform a query, the user calls the *expose*(v, w) operation, which returns a root cluster having v and w as endpoints (or *null*, if v and w are in different trees). Note that, even if v and w are in the same tree, *expose* may need to change the contraction to ensure that the root cluster actually represents the path from v to w. In principle, the user should define the internal functions so that the cluster returned by *expose* automatically contains the answer to the user's query; there is no need to actually search the tree. Top trees support aggregation over paths or trees directly, with no degree limitation. In particular, they naturally support applications that require both types of aggregation, such as maintaining the diameter, the center, or the median of a tree [6].

The first contraction-based data structures were in fact not top trees but Frederickson's *topology trees* [11,12,13]. They interpret clusters as *vertices* instead of edges. This leads to a simpler contraction algorithm, but it requires all vertices in the forest to have degree bounded by a constant. Although ternarization can remedy this, it is somewhat inelegant and adds an extra layer of complexity to the data structure. Recently, Acar et al. [1,2] invented *RC-trees*, which can be seen as a simpler, randomized version of topology trees. RC-trees also require the underlying tree to have bounded degree, and use space proportional to the bound (following [2], we set the bound to 8 in our experiments). We call the implementation (by Acar et al. [2]) of RC-trees used in our experiments RC. The

name "RC-trees" is a reference to *rake* and *compress*, which were introduced by Miller and Reif [19] in the context of parallel algorithms.

RC-trees have a generic interface to separate value updates from the actual contraction. Unlike top trees, queries require a traversal of the tree. This makes queries faster, but requires extra implementation effort from the user and is less intuitive than a simple call to *expose* (as in top trees). In addition, the interface assumes that the underlying tree has bounded degree: with ternarization, the interface will be to the transformed tree, with dummy vertices.

Alstrup et al. [6] proposed implementing top trees not as a standalone data structure, but as a layer on top of topology trees. Given the complexity of topology trees, this extra layer (which may as much as double the depth of the contraction) is undesirable. Recently, Holm, Tarjan, Thorup and Werneck proposed a direct implementation that still guarantees $O(\log n)$ worst-case time without the extra layer of topology trees. (Details, still unpublished, can be found in [27].) The data structure represents a natural contraction scheme: it works in rounds, and in each round performs a maximal set of independent pairings (i.e., no cluster participates in more than one pair). Level zero consists of all base clusters (the original edges). Level $i+1$ contains *rake* and *compress* clusters with children at level i, with *dummy clusters* added as parents of unpaired level-i clusters.

After a *link*, *cut*, or *expose*, the contraction can be updated in $O(\log n)$ worst-case time [27]. In practice, however, this implementation (which we refer to as TOP-W) has some important drawbacks. First, to preserve the circular order, each level maintains a linked list representing an Euler tour of its clusters, which makes updating the contraction expensive. Second, even though a "pure" top tree representing an n-vertex forest will have no more than $2n$ nodes, when dummy nodes are taken into account this number might be as large as $6n$. As a result, TOP-W uses considerably more memory than simpler data structures.

To overcome these drawbacks, Tarjan and Werneck [26] proposed a self-adjusting implementation of top trees (which we call TOP-S) that supports all operations in $O(\log n)$ amortized time. It partitions a tree into maximal edge-disjoint paths, each represented as a *compress tree* (a binary tree of *compress* clusters with base clusters as leaves). Each subtree incident to a path is represented recursively as a binary tree of *rake* clusters (which is akin to ternarization, but transparent to the user), and its root becomes a middle child of a *compress* node. This is the same basic approach as ST-trees, but ST-trees represent vertex-disjoint paths, have no embedded ternarization, and do not support circular adjacency lists directly.

3 Experimental Results

Experimental Setup. This section presents an experimental comparison of the data structures discussed above: ET-trees (ET), self-adjusting top trees (TOP-S), worst-case top trees (TOP-W), RC-trees (RC), and ST-trees implemented both with splay trees (ST-V/ST-E) and explicitly (LIN-V/LIN-E). We tested these on algorithms for three problems: maximum flows, minimum spanning trees, and

shortest paths on arbitrary graphs. Given our emphasis on the underlying data structures, we did not test more elaborate dynamic graph algorithms, in which dynamic trees are typically just one of several components; the reader is referred to [28] for a survey on this topic.

All algorithms were implemented in C++ and compiled with `g++` 3.4.4 with the `-O4` (full optimization) option. We ran the experiments on a Pentium 4 running Microsoft Windows XP Professional at 3.6 GHz, 16 KB of level-one data cache, 2 MB of level-two cache, and 2 GB of RAM. With the exception of RC-trees, all data structures were implemented by the authors and are available upon request. RC-trees, available at *http://www.cs.cmu.edu/∼jvittes/rc-trees/*, were implemented by Acar, Blelloch, and Vittes [2]. We only tested RC-trees on online minimum spanning forests, readily supported by the code provided.

CPU times were measured with the `getrusage` function, which has precision of 1/60 second. We ran each individual computation repeatedly (within a single loop in the program) until the aggregate time (measured directly) was at least two seconds, then took the average. The timed executions were preceded by a single untimed run, used to warm up the cache. Running times do not include generating or reading the input data (which is done only once by the entire program), but include the time to allocate, initialize, and destroy the data structure (each done once per run within the program). For each set of parameters, we report the average results from five different randomized inputs.

To ensure uniformity among our implementations, we reused code whenever possible. In particular, routines for splaying were implemented only once (as template functions) and used by TOP-S, ST-E, ST-V, and ET. To update values, each data structure defines an inline function that is called by the splaying routine whenever there is a rotation. Also, the user-defined functions used by top trees were implemented as templates (thus allowing them to be inlined) and were shared by both top tree implementations. Values were stored as 32-bit integers. At initialization time, each data structure allocates all the memory it might need as a single block, which is managed by the data structure itself (the only exception is RC-trees, which allocates memory as needed in large blocks, and frees it all at once). All executions fit in RAM, unless specifically noted.

Maximum flows. One of the original motivations for dynamic tree data structures was the *maximum flow problem* (see, e.g., [3]). Given a directed graph $G = (V, A)$ (with $n = |V|$ and $m = |A|$) with capacities on the arcs, a *source* s and a *sink* t, the goal is to send as much flow as possible from s to t. We implemented the *shortest augmenting path* algorithm for this problem, due to Edmonds and Karp [10]. In each iteration, it finds a path with positive residual capacity that has the fewest arcs and sends as much flow as possible along it; the algorithm stops when no such *augmenting path* exists. Intuitively, the algorithm grows a path from s containing only admissible arcs (potential candidates to belong to the shortest path) until it reaches t, backtracking whenever it reaches a vertex with no outgoing admissible arcs. A direct implementation takes $O(n^2m)$ worst-case time, which can be reduced to $O(mn \log n)$ if we use dynamic trees to maintain a forest of admissible arcs. The modified algorithm always processes

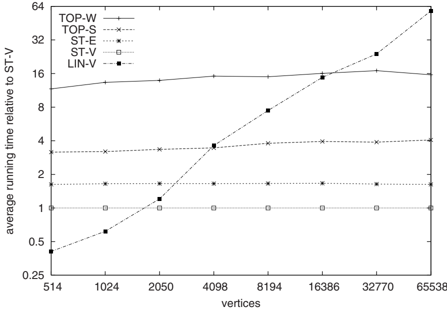


Fig. 1. Maximum flows on layer graphs with four rows and varying numbers of columns

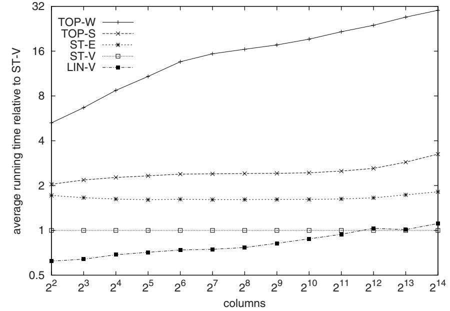


Fig. 2. Maximum flows on meshes with 65 538 vertices and varying numbers of columns (and rows)

the root v of the tree containing the source s. If v has an admissible outgoing arc, the arc is added to the forest (by *link*); otherwise, all incoming arcs into v are *cut* from the forest. Eventually s and t will belong to the same tree; flow is sent along the s-t path by decrementing the capacities of all arcs on the path and *cutting* those that drop to zero. See [3] for details.

The operations supported by the ST-tree interface (such as *addcost*, *findmin*, *findroot*) are, by construction, exactly those needed to implement this algorithm. With top trees, we make each cluster $C = (v, w)$ represent both a *rooted* tree and a *directed path* between its endpoints. The cluster stores the root vertex of the subtree it represents, a pointer to the minimum-capacity arc on the path between v and w (or *null*, if the root is neither v nor w), the actual capacity of this arc, and a "lazy" value to be added to the capacities of all subpaths of $v \cdots w$ (it supports the equivalent of ST-tree's *addcost*). We also tried implementing the full ST-tree interface on top of top trees, as suggested in [5], but it was up to twice as slow as the direct method.

Our first experiment is on *random layer graphs* [7], parameterized by the number of rows (r) and columns (c). Each vertex in column i has outgoing arcs to three random vertices in column $i + 1$, with integer capacities chosen uniformly at random from $[0; 2^{16}]$. In addition, the source s is connected to all vertices in column 1, and all vertices in column c are connected to the sink t (in both cases, the arcs have infinite capacity). We used $r = 4$ (thus making all augmenting paths have $\Theta(n)$ length) and varied c from 128 to 16 384. Figure 1 reports average running times normalized with respect to ST-V.

Our second experiment is on directed meshes, also parameterized by the number of rows (r) and columns (c). A mesh consists of a source s, a sink t, and an $r \times c$ grid. Each grid vertex has outgoing arcs to its (up to four) neighbors[1] with random integer capacities in the range $[0; 2^{16}]$. The grid is circular: the first and last rows are considered adjacent. In addition, there are infinite-capacity arcs

[1] A similar description was mistakenly given to the "directed meshes" tested in [27]; those graphs, obtained with a different generator [7], were actually acyclic.

from s to the first column, and from the last column to t. We kept the product of r and c constant at 2^{16} and varied their ratio. See Figure 2.

For both graph families, the $O(\log n)$ data structures have similar relative performance: ST-trees are the fastest, followed by self-adjusting top trees and, finally, worst-case top trees. Although there are costs (capacities) on edges, ST-V can be used because *evert* is never called; ST-E, included in the experiments for comparison, is slightly slower. With the linear-time data structure (LIN-V), the maximum flow algorithm is asymptotically worse, running in $O(n^2 m)$ time. Being quite simple, the algorithm is still the fastest with small trees, but is eventually surpassed by ST-V. On random layer graphs, this happens when augmenting paths have roughly 500 vertices; for directed meshes, both algorithms still have comparable running times when augmenting paths have more than 16 384 vertices. With so many columns, the average number of *links* between augmentations is almost 9000 on directed meshes, but only four on layer graphs (which are acyclic). This explains the difference in performance.

In the maximum flow algorithm, every query is soon followed by a structural update (*link* or *cut*). Next, we consider an application in which queries can vastly outnumber structural updates.

Online minimum spanning forests. The *online minimum spanning forest problem* is that of maintaining the minimum spanning forest (MSF) of an n-vertex graph to which m edges are added one by one. If we use a dynamic tree to maintain the MSF, each new edge can be processed in $O(\log n)$ time. Suppose a new edge $e = (v, w)$ is added to the graph. If v and w belong to different components, we simply add (*link*) e to the MSF. Otherwise, we find the maximum-cost edge f on the path from v to w. If it costs more than e, we remove (*cut*) f from the forest and add (*link*) e instead. This is a straightforward application of the "red rule" described by Tarjan in [24]: if an edge is the most expensive in *some* cycle in the graph, then it does not belong to the minimum spanning forest.

To find the maximum-cost edge of an arbitrary path with ST-trees, we simply maintain the negative of the original edge costs and call *findmin*. Because the *evert* operation is required, we must use ST-E in this case. With top trees, it suffices to maintain in each cluster $C = (v, w)$ a pointer to the maximum-cost base cluster on the path from v to w, together with the maximum cost itself.

Our first experiment is on random graphs: edges are random pairs of distinct vertices with integer costs picked uniformly at random from $[1; 1000]$. We varied n from 2^{10} to 2^{20} and set $m = 8n$. With this density, we observed that roughly 37% of the edges processed by the algorithm are actually inserted; the others generate only queries. Figure 3 shows the average time necessary to process each edge. For reference, it also reports the time taken by Kruskal's algorithm, which is *offline*: it sorts all edges (with quicksort) and adds them to the solution one at a time (using a union-find data structure to detect cycles). Being much simpler, it is roughly 20 times faster than ST-E.

Our second experiment also involves random graphs, but we now fix $n = 2^{16}$ and vary the average vertex degree from 4 to 512 (i.e., we vary m from 2^{17} to 2^{25}). As the density increases, relatively fewer *links* and *cuts* will be performed:

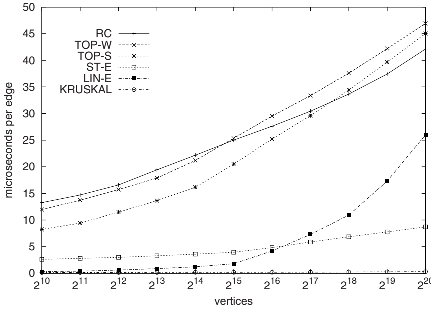


Fig. 3. Online minimum spanning forest on random graphs with average degree 16

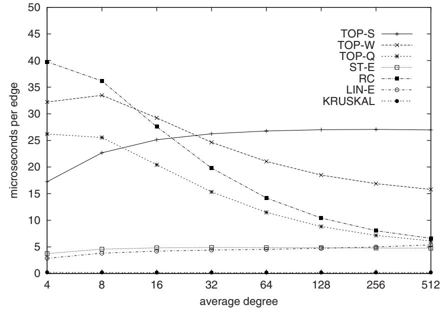


Fig. 4. Online minimum spanning forest on random graphs with 65 536 vertices

when the average degree is 512, only roughly 2.5% of the input edges are actually inserted into the MSF. As a result, as Figure 4 shows, the average time to process an edge decreases with the density, and queries dominate the running time. The speedup is more pronounced for TOP-W and RC, since the self-adjusting data structures (ST-E and TOP-S) must change the tree even during queries.

Recall that TOP-W must change the contraction when performing queries (*expose*) to ensure that the relevant path is represented at the root of its top tree. In principle, this would make *exposes* about as expensive as *links* and *cuts*. As suggested by Alstrup et al. [6], however, we implemented *expose* by marking some existing top tree nodes as "invalid" and building a temporary top tree with the $O(\log n)$ root clusters that remain. This eliminates expensive updates of Euler tours during queries. Fast queries help explain why TOP-W is more competitive with TOP-S for the online MSF application as compared to maximum flows. Being self-adjusting, TOP-S also benefits from the fact that consecutive dynamic tree operations are correlated in the maximum flow application.

RC-trees do not modify the tree (even temporarily) during queries: instead, they traverse the tree in a bottom-up fashion, aggregating information contained in internal nodes. For comparison, we have implemented TOP-Q, a variant of TOP-W that explicitly traverses the tree during queries, with no calls to *expose*. Technically, TOP-W is not an implementation of top trees, since it violates its well-defined interface. As Figure 4 shows, however, it is significantly faster than TOP-W when queries are numerous, and about as fast as RC. Speed comes at a cost, however: implementing a different query algorithm for each application is much more complicated (and less intuitive) than simply calling *expose*.

To further assess query performance, we tested the algorithms on *augmented random graphs*. A graph with n vertices, m edges, and *core size* $c \leq n$ is created in three steps: first, generate a random spanning tree on c vertices; second, progressively transform the original edges into paths (until there are n vertices in total); finally, add $m-n+1$ random edges to the graph. Costs are assigned so that only the first $n-1$ edges (which are processed first by the online MSF algorithm, in random order) result in *links*; the remaining edges result in queries only. Figure 5 shows the performance of various algorithms with $n = 2^{13}$, $m = 2^{18}$,

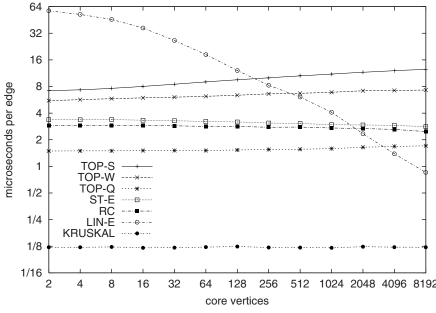


Fig. 5. Online minimum spanning forests on augmented random graphs

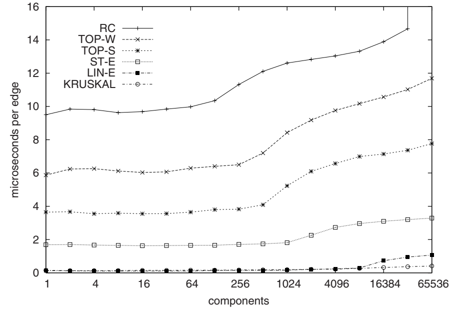


Fig. 6. Cache effects on forests with 32-vertex random components

and c varying from 2 to 2^{13}. The average length of the paths queried is inversely proportional to the core size; when the length drops below roughly 100, LIN-E becomes the fastest online algorithm (surpassing TOP-Q, which is particularly fast because more than 95% of the operations are queries). The crossover point between LIN-E and ST-E is closer to 150 (in Figure 3 as well).

Finally, we investigate the effect of caching on the data structures. We ran the MSF algorithm on graphs consisting of $32c$ vertices (for a given parameter c) randomly partitioned into c equal-sized components. Edges are inserted into the graph by first picking a random component, then a random pair of vertices within it. The total number of edges added is $128c$, so that each component has 128 edges on average. Figure 6 shows that, due to cache effects, the average time to process each edge actually varies as a function of c: all methods become slower as the number of components increases. Interestingly, LIN-E has the most noticeable slowdown: almost a factor of eight, compared to around two for other data structures. It benefits the most from caching when processing very small instances, since it has the smallest footprint per node (only 16 bytes). This is significantly less than ST-E (57 bytes), TOP-S (216), TOP-W (399), and RC (roughly one kilobyte). If fact, RC-trees even ran out of RAM for the largest graph tested (this is the only case reported in the paper in which this happened—all other tests ran entirely in memory); the excessive memory usage of this particular implementation helps explain why it is consistently slower than worst-case top trees, despite being presumably much simpler.

Even though Figure 6 shows an extreme case, cache effects should not be disregarded. Take, for instance, the layer graphs used in the maximum flow application. The graph generator assigns similar identifiers to adjacent vertices, which means that path traversals have strong locality. Randomly permuting vertex identifiers would slow down all algorithms, but LIN-V (which uses only 8 bytes per node) would be affected the most: on layer graphs with 65 538 vertices, running times would increase by 150% for LIN-V, 40% for ST-V (which uses 24 bytes per node), and only 11% for TOP-W.

Single-source shortest paths. The applications considered so far require dynamic trees to aggregate information over paths. We now test an application that

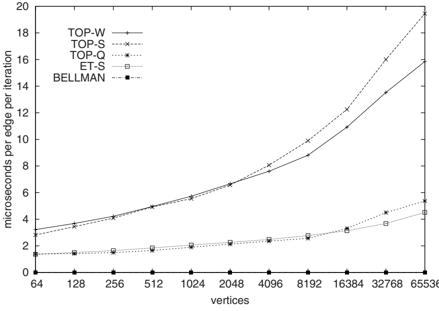


Fig. 7. Single-source shortest paths on graphs with Hamiltonian circuits

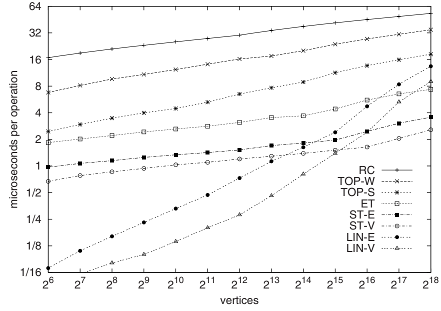


Fig. 8. Average time per operation on a randomized sequence of *links* and *cuts*

aggregates over trees: a label-correcting single-source shortest path algorithm. Given a directed graph $G = (V, A)$ (with $|V| = n$ and $|A| = m$), a length function ℓ, and a *source* $s \in V$, it either finds the distances between s and every vertex in V or detects a negative cycle (if present). Bellman's algorithm [8] maintains a *distance label* $d(v)$ for every vertex v, representing an upper bound on its distance from s (initially zero for s and infinite otherwise). While there exists an arc $(v, w) \in A$ such that $d(v) + \ell(v, w) < d(w)$, the algorithm *relaxes* it by setting $d(w) \leftarrow d(v) + \ell(v, w)$. After $n - 1$ passes through the list of arcs (in $O(mn)$ time) either all distance labels will be exact or a negative cycle will be found.

After an arc (v, w) is relaxed, we could immediately decrease the distance labels of all descendants of w in the current candidate shortest path tree. Bellman's algorithm will eventually do it, but it may take several iterations. With a dynamic tree data structure that supports aggregation over trees (such as ET-trees or top trees), we can perform such an update in $O(\log n)$ time. Although dynamic trees increase the worst-case complexity of the algorithm to $O(mn \log n)$, one can expect fewer iterations to be performed in practice.

We tested this algorithm on graphs consisting of a Hamiltonian circuit (corresponding to a random permutation of the vertices) augmented with random arcs. We used $m = 4n$ arcs, n of which belong to the Hamiltonian circuit. All arcs have lengths picked uniformly at random; those on the cycle have lengths in the interval $[1; 10]$, and the others have lengths in $[1; 1000]$.

Figure 7 shows the average time each method takes to process an arc. (ST-tree implementations are omitted because they cannot aggregate information over arbitrary trees.) ET-trees are much faster than both versions of top trees (TOP-W and TOP-S), and about as fast as TOP-Q, which explicitly traverses the tree during queries instead of calling *expose*. In these experiments, only 10% of the arcs tested result in structural updates. This makes TOP-W competitive with TOP-S, and TOP-Q competitive with ET (which is much simpler).

The standard version of Bellman's algorithm (denoted by BELLMAN), which maintains a single array and consists of one tight loop over the arcs, can process an arc up to 600 times faster than ET. Even though dynamic trees do reduce the number of iterations, they do so by a factor of at most four (for $n = 262\,144$, the algorithm requires 14.4 iterations to converge with dynamic trees and 56.6

without). As a result, ET is 100 to 200 times slower than Bellman's algorithm. This is obviously a poor application of dynamic trees, since the straightforward algorithm is trivial and has better running time; the only purpose of the experiment is to compare the performance of the data structures among themselves.

Random structural operations. In order to compare all data structures at once, we consider a sequence of m operations consisting entirely of *links* and *cuts*, with no queries. We start with $n-1$ *links* that create a random spanning tree. We then execute a sequence of $m-n+1$ alternating *cuts* and *links*: we remove a random edge from the current tree and replace it with a random edge between the two resulting components. We fixed $m = 10n$ and varied n from 2^6 to 2^{18}. For implementations of the ST-interface, every *link* or *cut* is preceded by the *evert* of one of the endpoints. Even though there are no queries, values were still appropriately updated by the data structure (as if we were maintaining the MSF). Figure 8 shows the average time to execute each operation of the precomputed sequence. The results are in line with those observed for previous experiments. ST-trees are the fastest logarithmic data structure, followed by ET-trees, self-adjusting top trees, worst-case top trees, and RC-trees.

Additional observations. An efficient implementation of the *evert* operation in ST-trees requires each node to store a *reverse bit* (in difference form), which implicitly swaps left and right children. Our implementation of LIN-V always supports *evert*, even in experiments where it is not needed (such as the maximum flow algorithm). Preliminary tests show that a modified version of LIN-V with no support for *evert* is roughly 5% faster in the maximum flow application. Also, as observed by Philip Klein (personal communication), an additional speedup of at least 10% can be obtained with a more specialized implementation of splaying that delays value updates until they are final (our current implementation does each rotation separately, updating all values). In an extreme case, if we do not update values at all during rotations, ST-V becomes almost 20% faster on a random sequence of *links* and *cuts*. The main reason is better locality: value updates require looking outside the splaying path.

The performance of the data structures also depends on how much data is stored in each node. If we stored values as 64-bit `doubles` (instead of 32-bit integers), all data structures would be slightly slower, but more compact ones would be affected the most. For random *links* and *cuts*, 64-bit values slow down ST-V by at least 10% and TOP-W by only 1%.

4 Final Remarks

We have shown that the linear-time implementation of the ST-tree interface can be significantly faster than other methods when the paths queried have up to a few hundred vertices, but they become impractical as path sizes increase. Alstrup et al. [5] observed the same for randomized sequences of operations. Recently, Ribeiro and Toso [21] have used the linear-time data structure as a building block for a simple method to maintain fully dynamic minimum spanning trees, which can be competitive with more elaborate algorithms for some graphs.

Among the logarithmic data structures, the self-adjusting implementation of ST-trees is generally the fastest, especially when *links* and *cuts* are numerous. It is relatively simple and can benefit from correlation of consecutive operations, as in the maximum flow application. Self-adjusting top trees are slower than ST-trees by a factor of up to four, but often much less. These are reasonably good results, given how much more general top trees are: our implementation supports sorted adjacency lists and aggregation over trees. As explained in [26], the data structure can be simplified if these features are not required (as in the maximum flow and MSF applications). We plan to implement restricted versions in the future, but even the current slowdown (relative to ST-trees) is arguably a reasonable price to pay for generality and ease of use. None of the logarithmic data structures studied is particularly easy to implement; the ability to adapt an existing implementation to different applications is a valuable asset.

When queries vastly outnumber *links* and *cuts*, worst-case and randomized data structures are competitive with self-adjusting ones. Even TOP-W, which changes the tree during queries, can be faster than self-adjusting top trees. But TOP-Q and RC prove that not making changes at all is the best strategy. Similar results were obtained by Acar et al. [2], who observed that RC-trees are significantly slower than ST-trees for structural operations, but faster when queries are numerous. In [9], a randomized implementation of ET-trees is used in conjunction with (self-adjusting) ST-trees to speed up connectivity queries within a dynamic minimum spanning tree algorithm. Although ST-trees can easily support such queries, the authors found them too slow.

This situation is not ideal. A clear direction for future research is to create general data structures that have a more favorable trade-off between queries and structural operations. A more efficient implementation of worst-case top trees would be an important step in this direction. In addition, testing the data structures on more elaborate applications would be valuable.

Acknowledgements. We thank Umut Acar, Guy Blelloch, Adam Buchsbaum, Jakob Holm, Philip Klein, Mikkel Thorup, and Jorge Vittes for helpful discussions.

References

1. Acar, U.A., Blelloch, G.E., Harper, R., Vittes, J.L., Woo, S.L.M.: Dynamizing static algorithms, with applications to dynamic trees and history independence. In: Proc. 15th SODA, pp. 524–533 (2004)
2. Acar, U.A., Blelloch, G.E., Vittes, J.L.: An experimental analysis of change propagation in dynamic trees. In: Proc. 7th ALENEX, pp. 41–54 (2005)
3. Ahuja, R., Magnanti, T., Orlin, J.: Network Flows: Theory, algorithms, and applications. Prentice-Hall, Englewood Cliffs (1993)
4. Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M.: Minimizing diameters of dynamic trees. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 270–280. Springer, Heidelberg (1997)
5. Alstrup, S., Holm, J., Thorup, M.: On the power and speed of top trees. Unpublished manuscript (1999)

6. Alstrup, S., Holm, J., Thorup, M., de Lichtenberg, K.: Maintaining information in fully dynamic trees with top trees. ACM TALG 1(2), 243–264 (2005)
7. Anderson, R.: The washington graph generator. In: Johnson, D.S., McGeoch, C.C.: (eds.), DIMACS Series in Discrete Mathematics and Computer Science, pp. 580–581. AMS (1993)
8. Bellman, R.: On a routing problem. Quarterly Mathematics 16, 87–90 (1958)
9. Cattaneo, G., Faruolo, P., Ferraro-Petrillo, U., Italiano, G.F.: Maintaining dynamic minimum spanning trees: An experimental study. In: Paliouras, G., Karkaletsis, V., Spyropoulos, C.D. (eds.) Machine Learning and Its Applications. LNCS (LNAI), vol. 2049, pp. 111–125. Springer, Heidelberg (2002)
10. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. JACM 19, 248–264 (1972)
11. Frederickson, G.N.: Data structures for on-line update of minimum spanning trees, with applications. SIAM J. Comp. 14(4), 781–798 (1985)
12. Frederickson, G.N.: Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. SIAM J. Comp. 26(2), 484–538 (1997)
13. Frederickson, G.N.: A data structure for dynamically maintaining rooted trees. J. Alg. 24(1), 37–65 (1997)
14. Goldberg, A.V., Grigoriadis, M.D., Tarjan, R.E.: Use of dynamic trees in a network simplex algorithm for the maximum flow problem. Mathematical Programming 50, 277–290 (1991)
15. Henzinger, M.R., King, V.: Randomized fully dynamic graph algorithms with polylogarihmic time per operation. In: Proc. 29th STOC, pp. 519–527 (1997)
16. Kaplan, H., Molad, E., Tarjan, R.E.: Dynamic rectangular intersection with priorities. In: Proc. 35th STOC, pp. 639–648 (2003)
17. Klein, P.N.: Multiple-source shortest paths in planar graphs. In: Proc. 16th SODA, pp. 146–155 (2005)
18. Langerman, S.: On the shooter location problem: Maintaining dynamic circular-arc graphs. In: Proc. 12th CCCG, pp. 29–35 (2000)
19. Miller, G.L., Reif, J.H.: Parallel tree contraction and its applications. In: Proc. 26th FOCS, pp. 478–489 (1985)
20. Radzik, T.: Implementation of dynamic trees with in-subtree operations. ACM JEA, vol. 3(9) (1998)
21. Ribeiro, C.C., Toso, R.F.: Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes on edge weights. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 393–405. Springer, Heidelberg (2007)
22. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. JCSS 26(3), 362–391 (1983)
23. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. JACM 32(3), 652–686 (1985)
24. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM Press (1983)
25. Tarjan, R.E.: Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. Mathematical Programming 78, 169–177 (1997)
26. Tarjan, R.E., Werneck, R.F.: Self-adjusting top trees. In: Proc. 16th SODA, pp. 813–822 (2005)
27. Werneck, R.F.: Design and Analysis of Data Structures for Dynamic Trees. PhD thesis, Princeton University (2006)
28. Zaroliagis, C.D.: Implementations and experimental studies of dynamic graph algorithms. In: Fleischer, R., Moret, B., Schmidt, E.M. (eds.) Experimental Algorithms: From Algorithm Design to Robust and Efficient Software. LNCS, pp. 229–278. Springer, Heidelberg (2002)

On the Cost of Persistence and Authentication in Skip Lists*

Michael T. Goodrich[1], Charalampos Papamanthou[2], and Roberto Tamassia[2]

[1] Department of Computer Science, University of California, Irvine
[2] Department of Computer Science, Brown University

Abstract. We present an extensive experimental study of authenticated data structures for dictionaries and maps implemented with skip lists. We consider realizations of these data structures that allow us to study the performance overhead of authentication and persistence. We explore various design decisions and analyze the impact of garbage collection and virtual memory paging, as well. Our empirical study confirms the efficiency of authenticated skip lists and offers guidelines for incorporating them in various applications.

1 Introduction

A proven paradigm from distributed computing is that of using a large collection of widely distributed computers to respond to queries from geographically dispersed users. This approach forms the foundation, for example, of the DNS system. Of course, we can abstract the main query and update tasks of such systems as simple data structures, such as distributed versions of dictionaries and maps, and easily characterize their asymptotic performance (with most operations running in logarithmic time). There are a number of interesting implementation issues concerning practical systems that use such distributed data structures, however, including the additional features that such structures should provide. For instance, a feature that can be useful in a number of real-world applications is that distributed query responders provide *authenticated responses*, that is, answers that are provably trustworthy. An authenticated response includes both an answer (for example, a yes/no answer to the query "is item x a member of the set S?") and a proof of this answer, equivalent to a digital signature from the data source.

Given the sensitive nature of some kinds of data and the importance of trustworthy responses to some kinds of queries, there are a number of applications of authenticated data structures, including scientific data mining [3, 13, 16, 17], geographic data servers [10], third party data publication on the Internet [5], and certificate revocation in public key infrastructure [1, 4, 7, 11, 14, 20, 21].

* This work was supported in part by IAM Technology, Inc. and by the National Science Foundation under grants IIS–0324846 and CCF–0311510. The work of the first author was done primarily as a consultant to Brown University.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 94–107, 2007.
© Springer-Verlag Berlin Heidelberg 2007

For example, in the CalSWIM project (see calswim.org), with whom we are collaborating, data providers allow users to query their water monitoring data and create publishable summaries from such queries. Because of the high stakes involved, however, in terms of both health and politics, data providers and data users want to be assured of the accuracy of the responses returned by queries. Thus, data authentication is a critical additional feature of some data querying applications. Another feature that is also of use in such applications (e.g., for non-repudiation) is the efficient implementation of persistence [6], which allows users to ask questions of a previous version of the data structure. That is, a persistent data structure can provide answers to the queries "was item x a member of the set S at time t?". For example, in the CalSWIM application, a user might wish to compare the level of a given water constituent today with its level several months ago.

In this paper, we study the features of persistence and authentication, and report on results concerning various implementation choices. In particular, we study the relative costs of adding persistence and authentication to dictionaries and maps, answering the question of how much overhead is needed in order to provide this extra functionality that is called for in some applications (like CalSWIM). Also we investigate various implementation decisions (e.g., using different primitive data structures, such as arrays and pointers, to implement more complex data structures) concerning dictionaries and maps. We focus on implementations based on the skip list data structure [26], as this simple data structure has been proven empirically to be superior in practice to binary search trees. In addition, we also show results concerning more "system-oriented" implementation issues, such as the influence of the garbage collector in our experiments, some system limitations concerning scaling of the performance in terms of memory usage and also the effect of virtual memory paging on the performance.

Concerning related previous work, as an experimental starting point, we note that Pugh [26] presents extensive empirical evidence of the performance advantages of skip lists over binary search trees. To review, the *skip list* data structure is an efficient means for storing a set S of elements from an ordered universe. It supports the operations $\mathsf{find}(x)$ (determine whether element x is in S), $\mathsf{insert}(x)$ (insert element x in S) and $\mathsf{delete}(x)$ (remove element x from S). It stores a set S of elements in a series of linked lists S_0, S_1, S_2,..., S_t. The base list, S_0, stores all the elements of S in order, as well as sentinels associated with the special elements $-\infty$ and $+\infty$. Each successive list S_i, for $i \geq 1$, stores a sample of the elements from S_{i-1}. To define the sample from one level to the next, we choose each element of S_{i-1} at random with probability $\frac{1}{2}$ to be in the list S_i. The sentinel elements $-\infty$ and $+\infty$ are always included in the next level up, and the top level, t, is maintained to be $O(\log n)$ and contains only the sentinels w.h.p. We therefore distinguish the node of the top list S_t storing $-\infty$ as the *start node* s. An element that exists in S_{i-1} but not in S_i is said to be a *plateau* element of S_{i-1}. An element that is in both S_{i-1} and S_i is said to be a *tower* element in S_{i-1}. Thus, between any two tower elements, there are some plateau elements. Pugh advocated an *array-based* implementation of skip lists [26], in C, and this

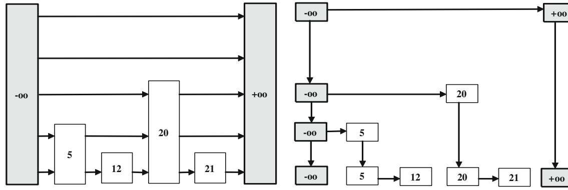


Fig. 1. Implementation of a skip list with arrays (left) and pointers (right)

is also the skip-list implementation used in the LEDA C++ library [19]. In this implementation, each element x in the set is represented with an array A_x of size equal to the height of the tower that corresponds to x. Each entry, $A_x[i]$, of the array A_x holds local data (for example in the authenticated data structures we store authentication information that correspond to $A_x[i]$) and a pointer to the array A_y, such that for all z such that $x < z < y$, the size of the array A_z is less than i. All the usual dictionary operations take time $O(\log n)$ with high probability and run quite fast in practice as well.

Using arrays for the towers is not the only way to implement skip lists, however. For example, there is an alternative pointer-based implementation of skip lists that uses nodes linked by pointers (see, e.g., [9]). We can optimize this implementation further, in fact, by taking advantage of the observation that for the operations supported by the skip list, some of the nodes and links are unnecessary and can be omitted. Then the skip list is somewhat like a binary tree (whose root is the highest-level node with value $-\infty$). In Figure 1 we illustrate two different implementations of a skip list storing the set $S = \{5, 12, 20, 21\}$, one array-based and the other pointer-based.

In addition to other studies of skip lists themselves (e.g., see [12, 15, 24, 25]), a considerable amount of prior work exists on authenticated data structures (e.g., see [2, 5, 7, 8, 10, 11, 20, 21, 27, 28]). In [8], the implementation of an authenticated dictionary with skip lists and commutative hashing is presented, again using arrays to implement the towers, but now using Java as the implementation language. In [2], a persistent version of authenticated dictionary is presented, where the user can now validate the historic membership of an element in a set, e.g., authenticated answers to the queries "was element e present in the data set S at time t?" Authenticated data structures for graph and geometric searching are presented in [10]. In [18], it is shown that almost any data structure can be converted into an authenticated one. In [22], alternate verification mechanisms are discussed. The authentication of simple database queries is studied in [23].

The main contribution of this paper is the empirical study of a number of important issues regarding the implementation of authenticated data structures using skip lists. In particular, we address the following questions: (i) the overheads for authentication and/or persistence; (ii) the relative costs of these various features and implementation decisions (e.g., which is more expensive–authentication or persistence?); (iii) the relative costs between updates and queries in authenticated skip lists; (iv) the differences between implementing

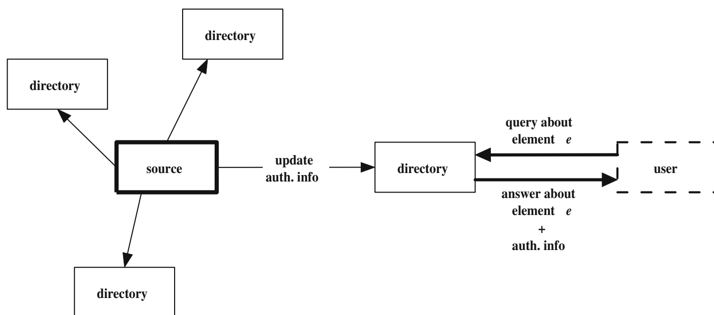


Fig. 2. Schematic illustration of an authenticated data structure

skip lists with arrays and pointers; (*v*) the impact of garbage collection on the performance (especially for fast $O(\log n)$-time operations); and (*vi*) the system limitations for implementing authenticated skip lists, including maximum memory constraints and the impacts of paging in virtual memory. We give extensive experiments that address all of the above questions. In addition, we show how to extend the notion of a persistent authenticated *dictionary* [2] to a persistent authenticated *map*, where each key is bound to a certain value and we present an efficient implementation of this new feature.

2 Implementing Authenticated Dictionaries and Maps

An authenticated data structure involves three types of parties: a trusted *source*, one or more untrusted *directories*, and one or more *users* (see Figure 2). The *source* maintains the original version of the data structure S by performing insertions and deletions over time. A *directory* maintains a copy of the data structure and periodically receives time-stamped updates from the source plus signed statements about the updated version of the data structure S. A user contacts a directory and performs membership queries on S of the type "is element e present in S?" (authenticated dictionary) or of the type "was element e present in S at time t?" (persistent authenticated dictionary) or finally of the type "what value was element e bound to at time t?" (persistent authenticated map). The contacted directory returns to the user an *authenticated response*, which consists of the answer to the query together with the *answer authentication information*, which yields a cryptographic proof of the answer assembled by combining statements signed by the source with data computed by the directory. The user then verifies the proof by relying solely on its trust in the source and the availability of public information about the source that allows the user to check the source's signature. The design of authenticated data structures should address the following goals:

- *Low computational cost:* The computations performed internally by each entity (source, directory, and user) should be simple and fast.

- *Low communication overhead:* source-to-directory communication (update authentication information) and directory-to-user communication (answer authentication information) should be kept as small as possible.
- *High security:* the authenticity of the answers given by a directory should be verifiable with a high degree of certainty.

Regarding the computational cost, an authenticated data structure should efficiently support the following operations: updates (insertions and deletions) executed by the source and directories, queries executed by directories, and verifications executed by the users. Various efficient realizations of authenticated data structures have been designed that achieve $O(\log n)$ query update, and verification time on a dictionary. In this paper, we consider authenticated data structures based on skip lists [2, 8, 28].

2.1 Authenticated Dictionary

We consider an *authenticated dictionary* for a set S with n items consists of the following components:

- A skip list data structure storing the items of S.
- A collection of values $f(v)$ that label each node v of the skip list.
- A statement signed by the source consisting of the timestamp of the most recent label $f(s)$ of the start node of the skip list. We recall that s is the left-uppermost node of the skip list.

Let h be a commutative cryptographic hash function [8]. We recall that if g is a collision resistant cryptographic hash function[1] then the respective commutative cryptographic hash function h is defined as [8]

$$h(x, y) = g(\min\{x, y\}, \max\{x, y\})$$

We use h to compute the label $f(v)$ of each node v in the skip list, except for the nodes associated with the sentinel value $+\infty$. These labels are set to 0. The value $f(s)$ stored at the start node, s, represents a digest of the entire skip list. Intuitively, each label $f(v)$ accumulates the labels of nodes below v, possibly combined with the labels of some nodes to the right of v.

We present this computation by using the pointer-based implementation,[2] where each node v stores two pointers, $u = \mathsf{down}(v)$ and $w = \mathsf{right}(v)$. We distinguish the following cases:

- if v is on the base level, then $f(v) = h(\mathsf{elem}(v), x)$, where x is the element of the node following v on the base level (this node may or may not have a right pointer from v);

[1] This means that it is computationally infeasible to find two inputs x_1 and x_2 such that $g(x_1) = g(x_2)$ and to invert the function.

[2] Note that the implementation of an authenticated skip list mentioned in [8] is array-based.

– if v is not on the base level, then
  - if $w = \mathsf{null}$, then $f(v) = f(u)$.
  - if $w \neq \mathsf{null}$ (w is a plateau node), then $f(v) = h(f(u), f(w))$.

When we update an authenticated dictionary we need to take into account the labels of the nodes. Consider an insertion (see Figure 3(a)). We start by performing an insertion into the underlying skip list. Next, we recompute the affected labels of the nodes of the skip list as well. It turns out that all we have to do is to update the labels of the nodes belonging to the path P that we traverse when we search for the element we want to insert. The length of this path is $O(\log n)$ with high probability. The importance of the authenticated data structures comes in the verification of the result. When the user queries the directory about the membership of an element x in the dictionary, the directory returns the answer and the answer authentication information, which consists of the signed digest that was received from the source and a sequence of values stored along the search path for x (see Figure 3(a)). This sequence has length $O(\log n)$. The users verifies first the signature of the signed digest. If this verification succeeds, the user recomputes the digest, by iteratively hashing the sequence of values, and compares this computed digest with the signed digest. For more details on authenticated data structures and their implementation with skip lists, see [8, 2].

2.2 (Persistent) Authenticated Maps and Dictionaries

An *authenticated map* data structure is an authenticated data structure where each key k is bound to a certain value v. The authentication information in this case should also take into account the association between keys and values.

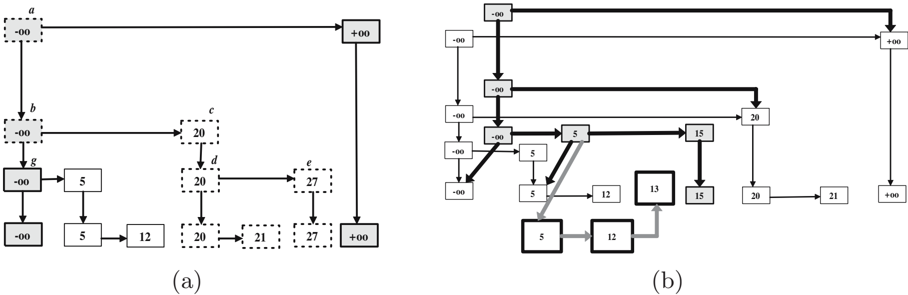


Fig. 3. (a) Insertion of element 27 in the skip list of Figure 1. The labels (hash values) of the dotted nodes change. The answer authentication information for a query on element 21 is the sequence $(27, 21, 20, f(e), f(g))$. (b) The insertion of element 15 creates a new version of the persistent authenticated dictionary. The insertion of element 13 is made next on this version, without creating a new version. Note that in this case we copy only the nodes 5, 12 (that were originally created when the very first version was instantiated) and create the large nodes 5 and 12. Note that in the final version. the node with element 5 will have only the gray down pointer.

We can build an authenticated map on top of an authenticated dictionary by storing the value together with the key only at the nodes of the base level. The authentication information for the authenticated map is computed as follows: if v is on the base level, then $f(v) = h(h(\mathsf{elem}(v), \mathsf{value}(v)), h(x, y))$, where x and y are the key and value of the node following v on the base level (this node may or may not have a right pointer from v); else the authentication information is computed exactly in the same way as in the authenticated dictionary. Regarding the query authentication information, the proof returned by the directory also contains key-value pairs. Suppose, for example, the skip list of Figure 3(a) includes the (key,value) pairs $(27, \alpha)$, $(21, \beta)$ and $(20, \gamma)$. Then the query authentication information for a query on key 21 would be the sequence $\{(27, \alpha), (21, \beta), (20, \gamma), f(e), f(g)\}$.

Additionally, a *persistent authenticated dictionary* [2] is an authenticated dictionary that supports also queries on past versions of the data structure of the type "was element e present in the dictionary at time t?". We can implement a persistent authenticated dictionary by modifying the implementation of an authenticated dictionary such that every update (insertion or deletion) creates a new version of the data structure. As shown in [2], to implement this update, we only have to copy nodes that belong to the search path of the skip list. This is because the authentication information of only these nodes differs for the two successive versions of the data structure. In [2], the insertions/deletions always create a new version of the data structure. We have implemented a more versatile form of persistence that supports also updates to the current version of the data structure, without creating a new version. In Figure 3(b), we illustrate the insertion algorithm. The insertion of element 15 creates a new version of the data structure and then the insertion of element 13 is done on the same version. Note that during the latter insertion, we only have to copy nodes that were originally created when a previous version of the data structure was instantiated.

3 Experimental Results

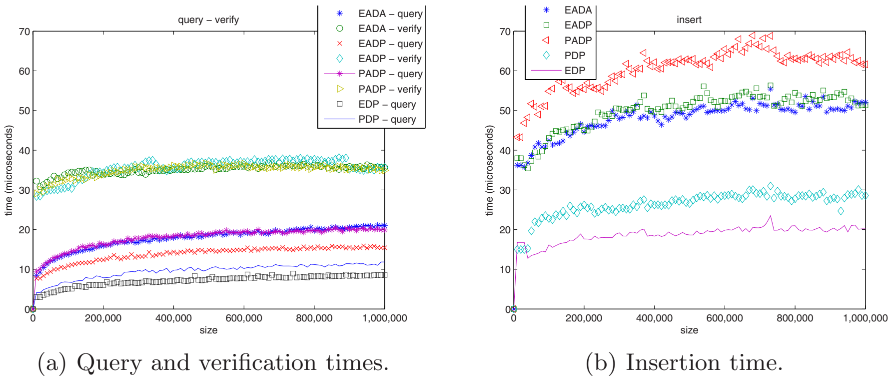
We have conducted experiments on the performance of various types of skip lists (ephemeral, persistent, and/or authenticated) containing up to 1,000,000 randomly generated 256-bit integers (for dictionaries) or pairs of two randomly generated 256-bit integers (for maps). For each operation, the average running time was computed over 10,000 executions. The experiments were conducted on a 64-bit, 2.8GHz Intel based, dual-core, dual processor machine with 8GB main memory and 2MB cache, running Debian Linux 3.1 with Linux kernel 2.6.15 and using the Sun Java JDK 1.5. The Java Virtual Machine (JVM) was most of the times launched with a 7GB maximum heap size. Cryptographic hashing was performed using the standard Java implementation of the MD5 algorithm.

We report the running times obtained in our experiments separating the time consumed by the garbage collector. Our experimental results provide a measure of the computational overhead caused by adding persistence and/or authentication to a dictionary implemented with skip lists. Also, they show some advantage

in relative performance for some operations (e.g., queries) between pointer-based and array-based implementations. The running times reported also exclude the time for signing the digest by the source and the time for verifying the signature on the digest by the user. Each of these times is about 5 milliseconds for 1,024 bit RSA signatures on MD5 (128 bit) digests using the standard java.security.∗ package. In typical applications, multiple updates are performed by the source during a time quantum. Thus, the signature creation time by the source should be amortized over all such update operations. Similarly, the user typically verifies multiple answers during a time quantum and thus the signature verification time should be amortized over all such verification operations.

3.1 Authenticated Dictionary

We first compare five different data structures based on the *dictionary* abstract data type and supporting yes/no membership queries: (a) *Ephemeral Authenticated Dictionary Array-based* (*EADA*), (b) *Ephemeral Authenticated Dictionary Pointer-based* (*EADP*), (c) *Persistent Authenticated Dictionary Pointer-based* (*PADP*), (d) *Persistent Dictionary Pointer-based* (*PDP*), (e) *Ephemeral Dictionary Pointer-based* (*EDP*). The results of the experiments are summarized in Figure 4, where we do not take into account the time taken by the garbage collector, which is discussed in Section 3.3. We discuss first the running times for pointer-based implementation and then consider array-based implementations.



(a) Query and verification times.

(b) Insertion time.

Fig. 4. Query, verification and insertion times (in microseconds) for ephemeral, persistent, and authenticated dictionaries implemented with skip lists, excluding the time for garbage collection. The times shown are the average of 10,000 executions on dictionary sizes varying from 0 to 10^6 elements.

Regarding queries (Figure 4(a)), the overhead due to persistence is less than the one due to authentication. See, e.g., the running times of PDP vs. EADP. This is explained by the fact that a query in an ephemeral authenticated dictionary has to assemble the proof by retrieving a logarithmic number of hash

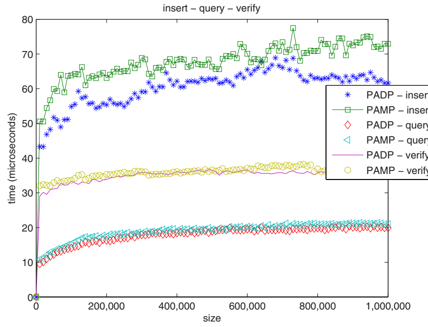
values located at nodes adjacent to the nodes encountered in the search (but does not perform any hash computation). Also, every query (whether unsuccessful or successful) must reach the bottom level of the skip list. Conversely, in a persistent (non authenticated) dictionary, a query does not have to assemble any proof and, if successful, may stop before reaching the bottom level. Adding both persistence and authentication (see the times for PADP) combines the two overheads, as expected. It should be also noted that the running time of verification operations (Figure 4(a)) does not depend on the data structure since in all cases, a verification consists of executing a sequence of cryptographic hash computations. The verification time is proportional to the size of the authentication information. Our experiments indicate that the number of values in the authentication information is about $1.5 \log n$, thus confirming the analysis in [28].

Regarding insertions (Figure 4(b)), we observe the same relative performance of the various data structures as for queries. However, the authentication overhead is now more significant since an update of an authenticated data structure requires a logarithmic number of cryptographic hash computations. Finally, we observe that for query operations array-based skip lists significantly underperform pointer-based ones. This is due to the fact that a query in the array-based implementation must traverse one by one all the levels, whereas in the pointer-based implementation, it can skip over multiple levels using pointers. In particular, the performance penalty of the array-based implementation is comparable to the authentication overhead, as can be seen in the running times for query operations in EADA vs. PADP. This can be explained by the fact that access to an array element in Java requires one level of indirection and that the JVM always checks that array indices are within bounds. For other operations, the array-based implementation has almost the same performance as the pointer-based implementation.

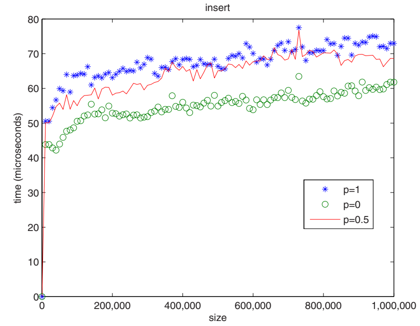
3.2 Authenticated Map

We also present experiments executed on persistent authenticated maps and dictionaries implemented with pointer-based skip lists (PAMP and PADP). The results of these experiments are summarized in Figure 5, where we do not take into account the time taken by the garbage collector, which is discussed in Section 3.3. Figure 5(a) compares the performance of persistent authenticated dictionaries with that of persistent authenticated maps. The experiments show that the additional functionality of the map abstract data type requires a small time overhead. For queries, the overhead is virtually nonexistent since searches in maps and dictionaries involve essentially the same computation. For insertion operations, the overhead is due to the need to perform some hash computations over larger data (the search key and its associated value).

In Figure 5(b) we present experiments on the running time of insertions in a variation of authenticated persistent maps such that a new version is created every $1/p$ insertions, where parameter p ($0 \leq p \leq 1$) denotes the frequency of insertions. For example, frequency $p = 0.25$ implies that 10,000 insertions cause



(a) Query, insertion and verification times.



(b) Insertion time in a persistent authenticated map for various version creation frequencies.

Fig. 5. Query, insertion, and verification times for persistent authenticated dictionaries and maps implemented with pointer-based skip lists, excluding the time for garbage collection. The times shown are the average of 10,000 executions on dictionary sizes varying from 0 to 10^6 elements.

the creation of 2,500 versions, frequency $p = 1$ denotes the original persistent authenticated map, and frequency $p = 0$ denotes an ephemeral map. Thus, in a series of m insertions, pm insertions create a new version while the remaining $(1 - p)m$ versions are executed on the current version. This variation of a persistent map models common applications where updates are accumulated and periodically executed in batches (e.g., daily expiration/revocation of access credentials). The chart of Figure 5(b) shows how the insertion time increases with the version creation frequency p.

Figure 6 summarizes the deletion time in persistent and ephemeral authenticated dictionaries and maps implemented with pointers. We can see that the deletion times are similar to the insertion times. Indeed, both delete and insert operations on the skip list involve an initial search, followed by pointers updates and recomputation of hash values along the search path.

3.3 Garbage Collector

In the charts of Figures 4–6, we have subtracted from the running times, the time consumed by the garbage collector (GC). The garbage collector is a thread of the JVM that periodically attempts to reclaim memory used by objects that will no longer be accessed by the running program.The exact schedule of the garbage collector cannot be completely controlled. Even if we force the garbage collector to run before and after a series of 10,000 operations (by explicitly calling System.gc()), we found that it also runs during the time interval that such operations are performed. Additionally, the garbage collector cannot not be turned off in the current version (JDK 1.5) of the Sun JVM. In our attempt

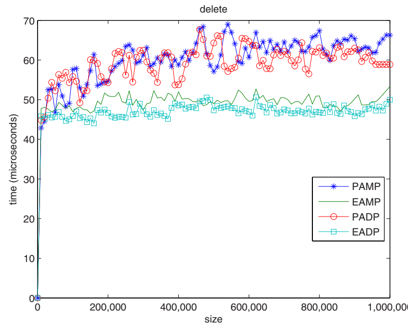
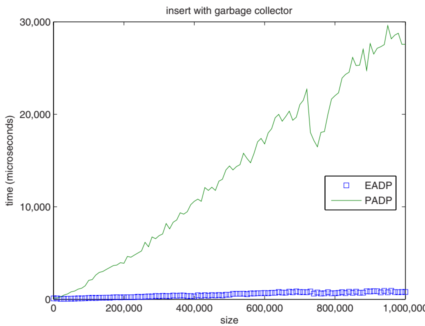
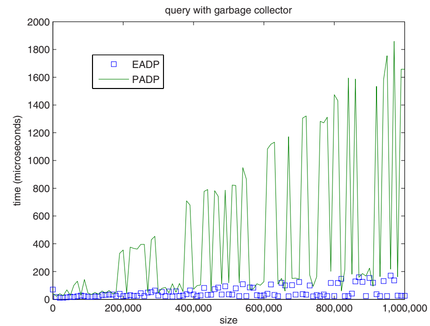


Fig. 6. Deletion times for authenticated (ephemeral and persistent) dictionaries and maps implemented with pointer-based skip lists, excluding garbage collection time



(a) Average insertion times, including garbage collection, over runs of 10,000 operations.



(b) Average query times, including garbage collection, over runs of 10,000 operations..

Fig. 7. The effect of the garbage collector on the insertion and query times on a persistent authenticated dictionary and an ephemeral authenticated dictionary

to reduce the execution of the garbage collector, we tried to tune it by using some JVM invocation options, such as

$$-\mathsf{XX} : +\mathsf{UseAdaptiveSizePolicy}, -\mathsf{XX} : \mathsf{MaxGCPauseMillis} = \mathsf{a}, -\mathsf{XX} : \mathsf{GCTimeRatio} = \mathsf{b},$$

that define the ratio of the execution times of the garbage collector and application. However we did not notice any important difference when we used the option $-\mathsf{XX} : \mathsf{MaxGCPauseMillis} = \mathsf{a}$ and we noticed only minor differences for the option $-\mathsf{XX} : \mathsf{GCTimeRatio} = \mathsf{b}$, such as a slightly lower frequency of execution of the garbage collector vs. the application, combined with more time per execution. Therefore, using these options did not influence the performance of the data structure. The behavior of the garbage collector is depicted in Figure 7.

The experiments show that the work of the garbage collector affects unevenly runs of 10,000 operations (both insertions (Figure 7(a)) and queries

(Figure 7(b)). Also the effect is more obvious in the case of the persistent authenticated dictionary, compared with the ephemeral authenticated dictionary. As the size of the data structure increases, the amount of time consumed by the garbage collector also increases. Also, since the frequency of garbage collection sweeps is constant over time, we have that the garbage collector overhead is greater for insertions (longer operations) than for queries (shorter operations).

3.4 System Limitations and Virtual Memory

In an attempt to see how our implementation scales up to large data sets, we have performed a series of insertions on the data structure until we get a Java out-of-memory exception. In this experiment, the JVM was launched with 8GB maximum heap size (we use all the memory we can before the system is forced to swap to the disk). Note that the JVM could have been launched with a lot more than 8GB maximum heap size since we use a 64-bit machine. In Figure 8(a), we can see how the version creation frequency p influences the exact size that can be handled by the JVM. As expected, the maximum size of the data structure decreases as the version creation frequency p grows. Note that for

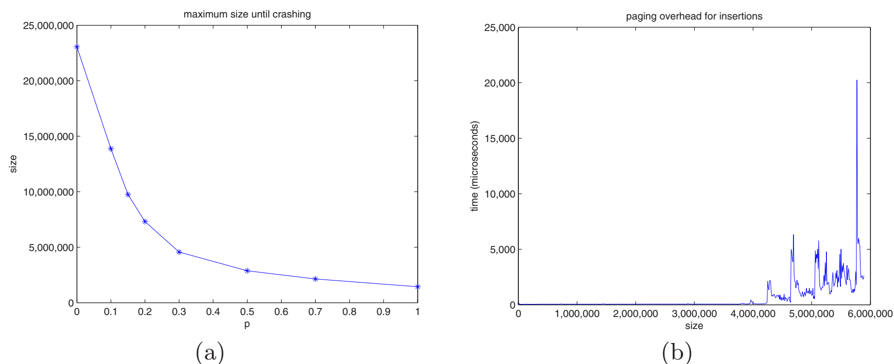


Fig. 8. (a) Maximum size (until machine crashes) of the persistent authenticated map as a function of the version creation frequency p. (b) Time overhead due to paging for insertions in a persistent authenticated map with version creation frequency $p = 0.5$.

$p = 0$ (ephemeral authenticated map), we can store up to 26 million items whereas for $p = 0.5$ we can store up to about 3 million items. Finally we show the influence of virtual memory on performance. To avoid an "out of memory" exception, we launch the JVM with more than 8GB maximum heap size so that the system will eventually resort to paging to disk. We show in Figure 8(b) how paging affects the performance of our persistent authenticated map. Namely, for $p = 0.5$, paging starts to impair performance when the map size reaches 3.5 million items.

4 Conclusions

In this paper, we present extensive experiments on authenticated data structures that are implemented with a skip list. We address implementation issues concerning ephemeral and persistent authenticated dictionaries and maps and we show that authenticated skip lists are quite efficient. We show that there are low overheads for adding authentication and persistence to distributed skip lists and extending authenticated dictionaries to become authenticated maps. We finally note that the overheads involved for garbage collection and virtual memory paging are not as significant as one might at first believe they would be.

Acknowledgments

The authors are grateful to Aris Anagnostopoulos for his contributions to an initial implementation of persistent authenticated dictionaries. We would like to thank also Nikos Triandopoulos and Danfeng Yao for useful discussions.

References

[1] Aiello, W., Lodha, S., Ostrovsky, R.: Fast digital identity revocation. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, Springer, Heidelberg (1998)

[2] Anagnostopoulos, A., Goodrich, M.T., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: Davida, G.I., Frankel, Y. (eds.) ISC 2001. LNCS, vol. 2200, pp. 379–393. Springer, Heidelberg (2001)

[3] Brunner, R.J., Csabai, L., Szalay, A.S., Connolly, A., Szokoly, G.P., Ramaiyer, K.: The science archive for the Sloan Digital Sky Survey. In: Proc. Astronomical Data Analysis Software and Systems Conference V (1996)

[4] Buldas, A., Laud, P., Lipmaa, H.: Eliminating counterevidence with applications to accountable certificate management. Journal of Computer Security 10(3), 273–296 (2002)

[5] Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.G.: Authentic data publication over the Internet. Journal of Computer Security 11(3), 291–314 (2003)

[6] Driscoll, J.R., Sarnak, N., Sleator, S., Tarjan, R.E.: Making data structures persistent. In: Proc. ACM Sympos. on the Theory of Computing, pp. 109–121 (1986)

[7] Gassko, I., Gemmell, P.S., MacKenzie, P.: Efficient and fresh certification. In: Imai, H., Zheng, Y. (eds.) PKC 2000. LNCS, vol. 1751, pp. 342–353. Springer, Heidelberg (2000)

[8] Goodrich, M.T., Tamassia, R.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: Proc. DARPA Information Survivability Conference & Exposition II (DISCEX II), pp. 68–82. IEEE Press, New York (2001)

[9] Goodrich, M.T., Tamassia, R.: Data Structures and Algorithms in Java, 4th edn. John Wiley & Sons, New York (2006)

[10] Goodrich, M.T., Tamassia, R., Triandopoulos, N., Cohen, R.: Authenticated data structures for graph and geometric searching. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 295–313. Springer, Heidelberg (2003)

[11] Gunter, C., Jim, T.: Generalized certificate revocation. In: Proc. 27th ACM Symp. on Principles of Programming Languages, pp. 316–329 (2000)

[12] Hanson, E.N.: The interval skip list: a data structure for finding all intervals that overlap a point. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1991. LNCS, vol. 519, pp. 153–164. Springer, Heidelberg (1991)

[13] Karp, R.M.: Mapping the genome: Some combinatorial problems arising in molecular biology. In: Proc. ACM Symp. on the Theory of Computing, pp. 278–285 (1993)

[14] Kaufman, C., Perlman, R., Speciner, M.: Network Security: Private Communication in a Public World. Prentice-Hall, Englewood Cliffs (1995)

[15] Kirschenhofer, P., Prodinger, H.: The path length of random skip lists. Acta Informatica 31, 775–792 (1994)

[16] Lupton, R., Maley, F.M., Young, N.: Sloan digital sky survey. http://www.sdss.org/sdss.html

[17] Lupton, R., Maley, F.M., Young, N.: Data collection for the Sloan Digital Sky Survey—A network-flow heuristic. Journal of Algorithms 27(2), 339–356 (1998)

[18] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica 39(1), 21–41 (2004)

[19] Mehlhorn, K., Näher, S.: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (2000)

[20] Micali, S.: Efficient certificate revocation. Technical Report TM-542b, MIT Laboratory for Computer Science (1996)

[21] Naor, M., Nissim, K.: Certificate revocation and certificate update. In: Proc. 7th USENIX Security Symposium, Berkeley pp. 217–228 (1998)

[22] Nuckolls, G.: Verified query results from hybrid authentication trees. In: DBSec, pp. 84–98 (2005)

[23] Pang, H., Tan, K.-L.: Authenticating query results in edge computing. In: Proc. Int. Conference on Data Engineering, pp. 560–571 (2004)

[24] Papadakis, T., Munro, J.I., Poblete, P.V.: Average search and update costs in skip lists. BIT 32, 316–332 (1992)

[25] Poblete, P.V., Munro, J.I., Papadakis, T.: The binomial transform and its application to the analysis of skip lists. In: Spirakis, P.G. (ed.) ESA 1995. LNCS, vol. 979, pp. 554–569. Springer, Heidelberg (1995)

[26] Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)

[27] Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003)

[28] Tamassia, R., Triandopoulos, N.: Computational bounds on hierarchical data processing with applications to information security. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 153–165. Springer, Heidelberg (2005)

Cache-, Hash- and Space-Efficient Bloom Filters

Felix Putze, Peter Sanders, and Johannes Singler

Fakultät für Informatik, Universität Karlsruhe, Germany
{putze,sanders,singler}@ira.uka.de

Abstract. A Bloom filter is a very compact data structure that supports approximate membership queries on a set, allowing false positives.

We propose several new variants of Bloom filters and replacements with similar functionality. All of them have a better cache-efficiency and need less hash bits than regular Bloom filters. Some use SIMD functionality, while the others provide an even better space efficiency. As a consequence, we get a more flexible trade-off between false positive rate, space-efficiency, cache-efficiency, hash-efficiency, and computational effort. We analyze the efficiency of Bloom filters and the proposed replacements in detail, in terms of the false positive rate, the number of expected cache-misses, and the number of required hash bits. We also describe and experimentally evaluate the performance of highly-tuned implementations. For many settings, our alternatives perform better than the methods proposed so far.

1 Introduction

The term *Bloom filter* names a data structure that supports membership queries on a set of elements. It was introduced already in 1970 by Burton Bloom [1]. It differs from ordinary dictionary data structures, as the result for a membership query might be true although the element is not actually contained in the set. Since the data structure is randomized by using hash functions, reporting a *false positive* occurs with a certain probability, called the *false positive rate (FPR)*. This impreciseness also makes it impossible to remove an element from a Bloom filter. The advantage of a Bloom filter over the established dictionary structures is space efficiency. A Bloom filter needs only a *constant number* of bits per (prospective) element, while keeping the FPR constant, *independent* from the size of the elements' universe.

The false positives can often be compensated for by recalculating or retransferring data. Bloom filters have applications in the fields of databases, network applications [2] and model checking [4,5]. The requirements on the Bloom filter and the way of usage differ greatly among these fields of applications.

Paper Outline. In Section 2 we review "standard" Bloom filters which are based on setting k bits in a bit array which are determined by k hash functions. Section 3 introduces and analyzes a family of cache-efficient variants of standard Bloom filters. There are two main ideas here: concentrate the k bits in one (or only few) cache blocks and precompute random bit patterns in order to save both

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 108–121, 2007.
© Springer-Verlag Berlin Heidelberg 2007

hash bits and access time. While these Bloom filter variants improve execution time at the cost of slightly increased FPR, the ones presented in Section 4 saves space by engineering practical variants of the theoretically space optimal Bloom filter replacements proposed by Pagh et. al. [11]. The basic idea is a compressed representation of a Bloom filter with $k = 1$. Our main algorithmic contribution is the observation that a technique from information retrieval fits perfectly here: Since the distances between set bits are geometrically distributed, Golomb codes yield almost optimal space [10]. After giving some hints on the implementation in Section 5, we present an experimental evaluation in Section 6. We conclude our paper in Section 7.

2 Standard Bloom Filters with Variants

The original Bloom filter for representing a set of at most n elements consists of a bit vector of length m. Let $c := m/n$ be the bits-per-element rate. Initially, all bits are set to 0. For inserting an element e into the filter, k bits are set in the Bloom filter according to the evaluations of k independent hash functions $h_1(e), \ldots, h_k(e)$. The membership query consists of testing all those bits for the query element. If all bits are set, the element is likely to be contained in the set, otherwise, it is surely not contained.

For a fixed number of contained elements, the FPR is lowest possible if the probability of a bit being set is as close as possible to $\frac{1}{2}$. One can easily show that it is optimal to choose $k = \ln 2 \cdot c = \ln 2 \cdot \frac{m}{n} \approx 0.693\frac{m}{n}$.

The probability that a bit has remained 0 after inserting n elements, is[1]

$$p' := \left(1 - \frac{1}{m}\right)^{kn} \overset{i=kn}{\approx} \lim_{i\to\infty} \left(1 - \frac{kn}{mi}\right)^{i} = \mathrm{e}^{-kn/m}\,. \tag{1}$$

The false positive rate for a standard Bloom filter (std) is

$$f_{\mathsf{std}}(m,n,k) = (1-p')^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \overset{!}{\approx} \frac{1}{2^k} \tag{2}$$

for the optimal k. The detailed calculation can be found in Mitzenmacher's survey paper [2].

Classification. The original Bloom filter can be termed a *semi-static* data structure, since it does not support deletions. Variants that *do* support deletion are called *dynamic*. The other extreme is a *static* filter where not even additions to the set may occur after construction.

Existing Variants for Different Requirements. A variant called *Counting Bloom Filters* [6] allows deletion of elements from the Bloom filter by using

[1] We assume throughout the paper that the hash functions are perfectly random.

(small) counters instead of a single bit at every position. This basic technique is also used by [11] in combination with a space-efficient multiset data structure, to yield an asymptotically space-optimal data structure.

In [7], Mitzenmacher et al. show that we can weaken the prerequisite of independent hash functions. The hash values can also be computed from a linear combination of two hash functions $h_1(e)$ and $h_2(e)$. This trick does not worsen the false positive rates in practice.

3 Blocked Bloom Filters

We will now analyze the cache efficiency of a standard Bloom filter, which we assume to be much larger than the cache. For negative queries, only less than two cache misses are generated, on the average. This is because each bit is set with probability $q = 1/2$, when choosing the optimal k, and the program will return false as soon as an unset bit is found. This cannot be improved much, since at most one cache fault is needed for accessing some randomly specified cell in the data structure.

Standard Bloom filters are *cache-inefficient* since k cache misses are generated by every input operation and (false or true) positive membership query.

In this section, we present a cache-efficient variant called *blocked Bloom filter* (blo). It consists of a sequence of b comparatively small standard Bloom filters (Bloom filter blocks), each of which fits into one cache-line. Nowadays, a common cache line size is 64 bytes = 512 bits. For best performance, those small filters are stored cache-line-aligned. For each potential element, the first hash value selects the Bloom filter block to be used. Additional hash values are then used to set or test bits as usual, but only inside this one block. A blocked Bloom filter therefore only needs one cache miss for every operation. In the setting of an external memory Bloom filter, the idea of blocking was already suggested in [8], but the increase of the FPR was found negligible for the test case there ($k = 5$), and no further analysis was done. The blocked Bloom filter scheme differs from the *partition schemes* mentioned in [7, Section 4.1], where each bit is inserted into a different block.

Let primed identifiers refer to the "local" parameters of the Bloom filter block. On the first glance, blocked Bloom filters should have the same FPR as standard Bloom filters of the same size since the FPR in Equation (2) only depends on k and n/m, since $k = k'$ and since the expected value of n'/m' is n/m. However, we are dealing with small values of m so that the approximation is not perfect. More importantly, n' is a random variable that fluctuates from block to block. Some blocks will be overloaded and others will be underloaded. The net effect is not clear on the first glance. The occupancies of the blocks follow a binomial distribution $B(n, 1/b)$ that can be closely approximated by a Poisson distribution with parameter $n/b = B/c$ since n is usually large, and B/c is a small constant.

An advantage of this approximation is that it is independent of the specific value of n. For the overall FPR of a blocked Bloom filter with local FPR $f_{\mathsf{inner}}(B, i, k)$ we get the following infinite but quickly converging sum:

$$f_{\mathsf{blo}}(B, c, k) := \sum_{i=0}^{\infty} \mathrm{Poisson}_{B/c}(i) \cdot f_{\mathsf{inner}}(B, i, k) \tag{3}$$

For a blocked Bloom filter using the typical value $c = 8$ bits per element, the decline in accuracy is not particularly bad; the FPR is 0.0231 instead of 0.0215 for $B = 512$ bits. By increasing c by one, we can (over-)compensate for that. For larger c, the effect of the non-uniform distribution can be diminished by choosing a smaller k than otherwise optimal. Still, for $c = 20$ and $k = 14$, the FPR almost triples: it rises from 0.0000671 to 0.000194, which might not be acceptable in certain applications. Thus, we have to increase c to 24. The numerically computed results for many values of c are shown in Table 1. These values are impractical for $c > 28$, since more than 50% additional memory must be used to compensate for the blocking. However, for $c < 20$, the additional memory required is only 20%. This can be acceptable, and often even comes with an improvement to the FPR, in the end. For $c > 34$, the blocked Bloom filter with $B = 512$ cannot compensate the FPR any more, for a reasonable number of bits per element.

Table 1. Increasing the space for a blocked Bloom filter to compensate the FPR (B=512)

c	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
c'	6	7	8	9	10	11	12	13	14	16	17	18	20	21	23	25	26	28	30	32	35	38	40	44	48	51	58	64	74	90
+%	20	16	14	12	11	10	9	8	7	14	13	12	17	16	21	25	23	27	30	33	40	46	48	57	65	70	87	100	124	165

Bit Patterns (pat). A cache miss is usually quite costly in terms of execution time. However, the advantage in performance by saving cache misses can still be eaten up if the computation is too complex. For the blocked Bloom filters, we still have to set or test k bits in for every insertion or positive query. On the other hand, modern processors have one or two SIMD units which can handle up to 128 bits in a single instruction. Hence, a complete cache-line can be handled in only two steps.

To benefit from this functionality, we propose to implement blocked Bloom filters using precomputed bit patterns. Instead of setting k bits through the evaluation of k hash functions, a single hash function chooses a precomputed pattern from a table of random k-bit pattern of width B. With this solution, only one small (in terms of bits) hash value is needed, and the operation can be implemented using few SIMD instructions. When transferring the Bloom filter, the table need not be included explicitly in the data, but can be reconstructed using the seed value.

The main disadvantage of the bit pattern approach is that two elements may cause a *table collision* when they are hashed to the same pattern. This leads to

an increased FPR. If ℓ is the number of table entries, the collision probability in an n element Bloom filter block is $p_{\text{coll}}(n, \ell) := 1 - \left(1 - \frac{1}{\ell}\right)^n$. Hence we can bound the FPR for one block by

$$f_{\mathsf{pat}}(m, n, k, \ell) \leq p_{\text{coll}}(\ell) + (1 - p_{\text{coll}}(\ell)) f_{\mathsf{std}}(m, n, k) . \tag{4}$$

This local FPR can be plugged into Equation (3) to yield the total FPR. Bit patterns work well when on the one hand, the pattern table is small enough to fit into the cache and on the other hand, the table is big enough to ensure that table collisions do not increase the FPR by too much.

Multiplexing Patterns. To refine this idea once more, we can achieve a larger variety of patterns from a single table by bitwise-or-ing x patterns with an average number of k/x set bits. Ignoring rounding problems, dependencies, etc.

$$f_{\mathsf{pat[x]}}(m, n, k, \ell) \approx f_{\mathsf{pat}}(m, xn, k/x, \ell)^x . \tag{5}$$

Multi-Blocking. One more variant that helps improving the FPR, is called *multi-blocking*. We allow the query operation to access X Bloom filters blocks, setting or testing k/X bits respectively in each block. (When k is not divisible by X, we set an extra bit in the first $k \bmod X$ blocks.) Multi-blocking performs better that just increasing the block size to XB, since more variety is introduced this way. If we divide the set bits among several blocks, the expected number of 1 bits per block remains the same. However, only k/X bits are considered in each participating block, when accessing an element. Thus, we have to generalize Equation (2):

$$f_{\mathsf{std[X]}}(m, n, k) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn/X}\right)^k \tag{6}$$

We get an estimate for the total FPR of

$$f_{\mathsf{blo[X]}}(B, c, k) := \sum_{i=0}^{\infty} \text{Poisson}_{XB/c}(i) \cdot f_{\mathsf{std[X]}}(B, i/X, k)^X \tag{7}$$

This can be adapted to bit patterns as before. The multiplexing and the multi-blocking factor will be denoted by appending them to either variant, i. e. blo[X] and pat[x, X] respectively.

Combinations. Using the formulas presented, all combinations of the variants presented here can be theoretically analyzed. Although the calculations make simplifying assumptions, mainly through disregarding dependencies, they match the experimental results closely, as shown in Figure 1. The differences are very small, and only appear for large c, where random noise in the experimental values comes into play.

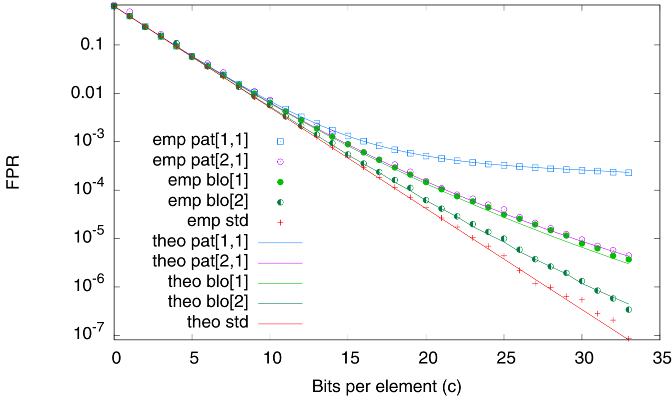


Fig. 1. Comparing the empirical FPR to the theoretically computed one, for some variants. The lines represent the theoretical, the points indicate the experimental results.

Table 2. Number of hash bits used by the various variants. f is the desired FPR, n the number of elements, and m the available space, B the block size, and ℓ the length of the pattern table. Throughout this paper, $\log x$ stands for $\log_2 x$.

Operation	Insert / Positive Query	Negative Query
std	$k \log m$	$2 \log m$
blo[X]	$X \log(m/B) + k \log B$	$\log(m/B) + 2 \log B$
pat[x, X]	$X(\log(m/B) + x \log \ell)$	$\log(m/B) + x \log \ell$
ch	$\log n/f$	$\log n/f$
gcs	$\log n/f$	$\log n/f$

Hash Complexity. Besides the cost for memory access, Bloom filters incur a cost for evaluating hash functions. Since the *time* needed for this is very application dependent, we choose to compare the different algorithms based on the number of *hash bits* needed for a filter access. Table 2 summarizes the results.

Exemplary values for $m = 800,000,000$ are shown in Figure 2. The values follow the theoretical computation in Table 2. Obviously, the proposed variants perform better than the standard Bloom filters, for a reasonable choice of c.

4 Space-Efficient Bloom Filter Replacements

In the previous section, we have proposed methods for making Bloom filters produce less cache faults and use less hash bits. The aim was to improve the execution time, while at the same time, sacrificing FPR and/or space efficiency. In this section, we describe Bloom filters with near optimal space consumption that are also cache-efficient. We pay for this with a trade-off between execution time and an additive term in the space requirement.

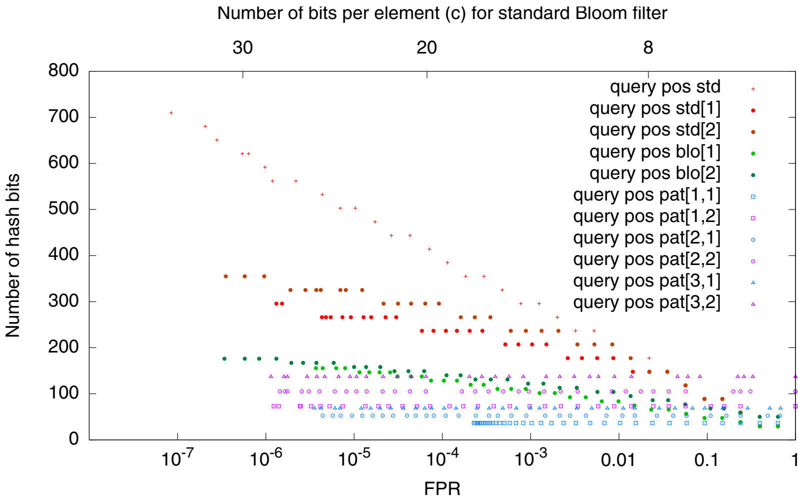


Fig. 2. Number of hash bits used against the FPR

Our basic solution is also static, i. e. the data structure must be constructed in a preprocessing step, with all elements given, before any query can be posed. At the end of this section we outline how the data structure can be dynamized.

The original Bloom filters are space-efficient, but not space-optimal [11]. When ignoring membership query time, one could just store one hash value in the range $\{1, \ldots, n/f\}$ to obtain an FPR of f. This would cost only $\log \binom{n/f}{n} \approx n \log \frac{e}{f}$ bits instead of $\frac{n}{\ln 2} \log(1/f)$ bits. Hence, a traditional Bloom filter needs about $1/\ln 2 \approx 1.44$ times as much space, even worse by an additive constant when compared to the information-theoretic minimum, $n \log(1/f)$ bits. This amount of extra memory can be saved by sacrificing some access time. Pagh and Pagh [11] use a asymptotically space-optimal hash data structure invented by Cleary [3] for storing just those hash values. Let this approach be termed *CH filter* (ch) here. However, to guarantee expected constant membership query time, a constant number of bits must be spent additionally for each contained element. Those bits comprise a structure that gives some hints to find the desired element more quickly. The more extra bits are provided, the faster the data structures will work. Although the number of bits is independent of n, and more importantly, of the FPR, it eats up most of the savings achieved, for reasonably small values of c. Another point is that a hash data structure should never get close to full, i. e. there must be some maximal load α, which in turn increases memory usage. Summarizing this, access time must be traded off with space efficiency again, but this time with the ability to get arbitrarily close to the theoretical optimum, asymptotically.

Our own solution proposed here is based on an approach used in search engines to store sorted lists of integers [12]. Imagine a simple Bloom filter with $k = 1$, i. e. a hashed bitmap, yielding an FPR of $1/c$. This bitmap can be greatly compressed, as the 1 bits are sparse. However, differently to [9], we do not use (optimal) arithmetic coding, since this prohibits random access (without un-

packing all the data again). Instead, we do not compress the bitmap, but the sorted sequence of hash values in the range $\{0, \ldots, nc\}$ for all the contained elements. These values are uniformly distributed, therefore, the differences between two successive values are geometrically distributed with $p = 1/c$. For a geometric distribution, *Golomb coding* [10, p. 36] is a minimal-redundancy code, i. e. by choosing the right parameter, we lose only at most half a bit per element, compared to the information-theoretic optimum.

However, this compressed sequence still does not allow any random-access, since the Golomb codes have value-dependent sizes. Hence, we have to augment it with a index data structure so we can seek to a place near the desired location quickly. Therefore, we divide the number range of the hash function into parts of equal size I. In addition, for each of these blocks, a bit-accurate pointer to the beginning of the subsequence that contains the corresponding values, is stored. So there is a trade-off once again: For a small search time we want a small I, but large I are good for saving space.

This data structure, termed *Golomb-Compressed Sequence* (gcs) is static, in contrast to the compact-hash approach, i.e., all hash values and thus, all elements in the set must be known beforehand.

Dynamization of gcs. We can support insertions by maintaining a small dynamic hash table T_i for recently inserted elements. It suffices if T_i stores the bit positions for the main table. When T_i becomes too big, we empty it by reconstructing the main table. With a bit of caution we can even support deletion using a deletion buffer T_d. This works if both the main table and T_d store multisets of bit positions. This can be done very space efficiently in the main table. We just need to provide a code word for the distance 0. Since this does not significantly increase the lengths of the other code words and since there are only few collisions, the resulting space and time overhead is small.

5 Implementation Aspects

Blocked Bloom filters with bit patterns profit from storing the Bloom filter in *negated* form—a membership query then reduces to testing whether the bitwise-and of the pattern and the negated filter block is zero. Insertion amounts to a bitwise-and of negated pattern and negated filter block.

To scale the hash values to the appropriate range, we use floating-point multiplication and rounding instead division with remainder. Our measurements indicate that this is crucial for performance.

We implemented all algorithms in a library-style way that makes them easy to use in real applications. This includes easy configuration of all tuning parameters, most of them allowed to be changed at runtime. Through generic programming, we are able to plug in arbitrary data types and hash functions, without any runtime overhead. The code can be obtained from the authors.

With all those details described, we can state that everything possible was done to achieve best practical performance for all of the contestants, thus guaranteeing a fair comparison.

6 Experimental Evaluation

We evaluate our implementations using one core of an Intel Xeon 5140 dual-core processor, running at 2.33 GHz with 4 MB of L2-cache using 64 Byte cache lines. They use the automatic vectorization feature of the Intel C++ Compiler[2] to enable SIMD operations without hand-written assembler code. We benchmark the operations

1. insert an element
2. query an element *contained* in the set, returning true
3. query an element *not contained* in the set, returning true or false

The elements are random strings of length 8. They are hashed using one or two variants of Jenkins' hash function [5] with different seeds which output a total of 64 bits. When even more bits are needed, hash values are generated as needed using a linear combination of those two hash values, as proposed in [7]. In each case, the number of elements n is chosen so that the filter data structure has size 95 MB. After inserting n elements, querying for the same set is performed. Additional n elements are queried for in the second step. This made it possible to measure the running times for both positive and negative queries. The cache-line size is 64 bytes, so we chose $B = 512$. For the pattern-based filters, the table size is set to the full 4 MB, resulting in $\ell = 64$K.

To make the comparison fair, we also include variants of the standard Bloom filter that for a given c use a k value just large enough to ensure an FPR at least as good as blo[1] (std[1]) and blo[2] (std[2]) respectively.

Figures 3 and 4 show running times for the positive and negative queries as well as the filter size, for c from 1 to 34. The insertion times are omitted since they are very similar to the times for positive queries.

As stated before, there is not much improvement to expect for negative queries, since their detection is already quite cache-efficient for the original Bloom filter. Also, they do not use many hash bits. For both positive queries and insertions, the blocked Bloom Filter variants outperform the original Bloom filter, in particular for low FPRs, i. e. high reliability. The maximum speedup factor is close to 4, using 32% more memory than the standard variant. However, the speedup is actually smaller than one would expect from the difference in cache misses (see Appendix A). Apparently, the system can hide cache latencies using prefetching.

The normal pattern variants are only slightly faster than the regular blocked Bloom filter. One reason is that we use very cheap hash functions. But there is another cause: When the pattern table occupies all of the cache, almost every filter access evicts a pattern from the cache whose next access will cause a cache fault. Reducing the table size (slowly) reduces this problem. The normal pattern variant also does not reach the area of very large FPRs. For $c = 34$, the FPR is limited by the probability of table collisions, about $512/34/64K \approx 2.3 10^{-4}$.

[2] Version 9.1.045.

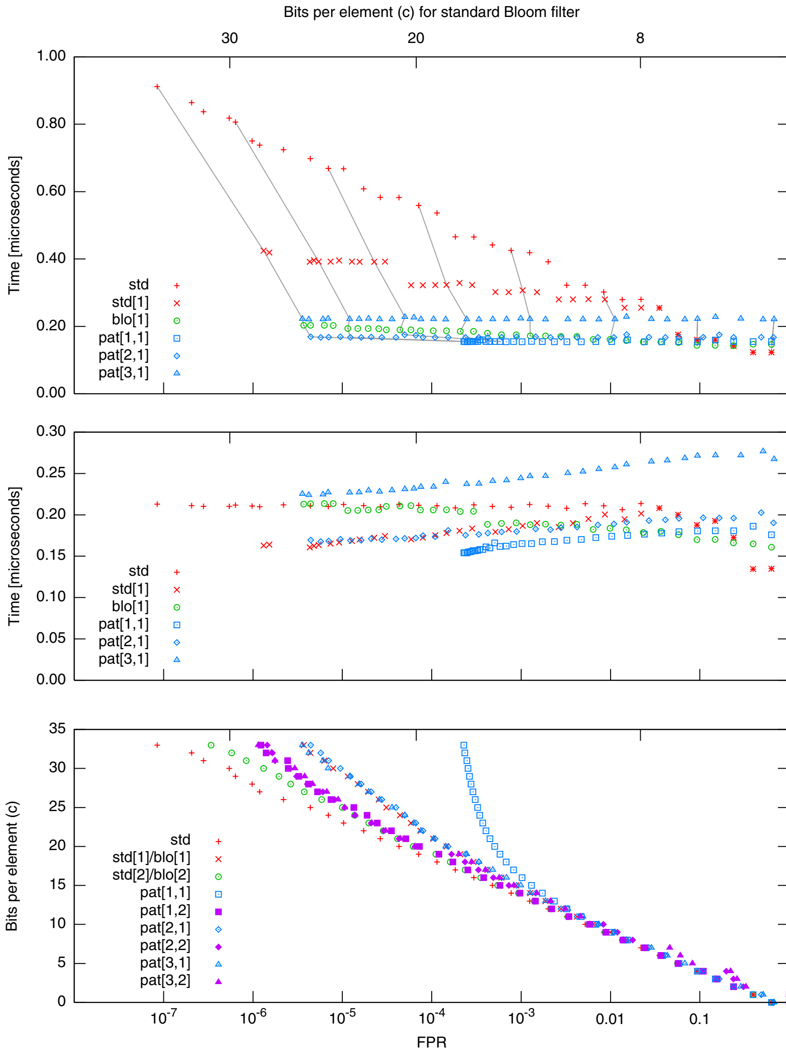


Fig. 3. Execution times for positive (top), negative (middle) queries, and size of filter in bits per contained element (bottom). For comparison, lines connect data points with the same number of bits per element. For readability, only the variants accessing one block are shown here in (top) and (middle), the two-block variants can be found in Figure 4.

The multiplexing versions are able to overcome this limitation, but need more computational effort.

Regarding the single-blocking variants, for positives and insertions, pat[1, 1] performs best for rather high FPRs, while pat[2, 1] extends this behavior to smaller FPRs, using 2-fold multiplexing.

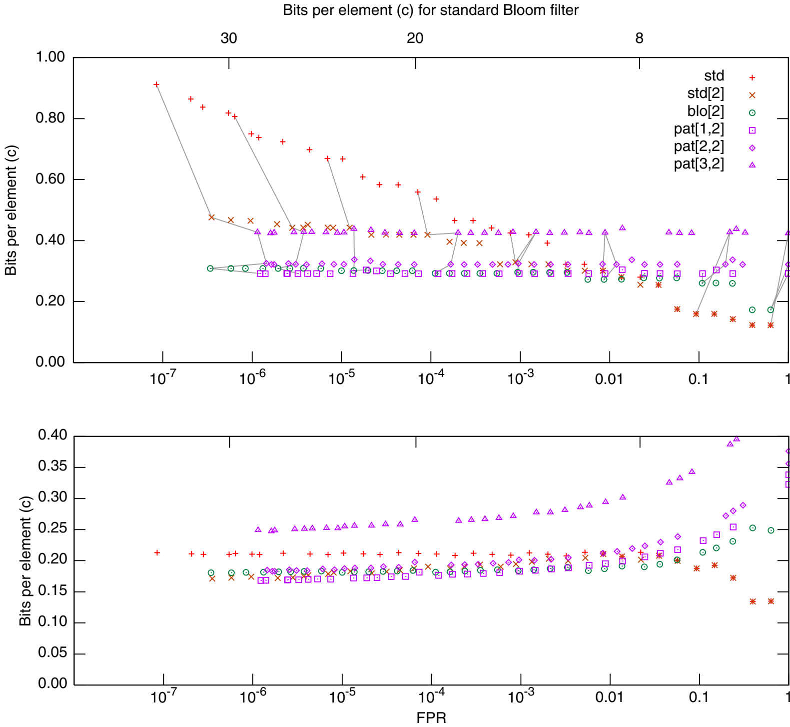


Fig. 4. Execution times for positive (top) and negative (bottom) queries for the variants accessing two blocks

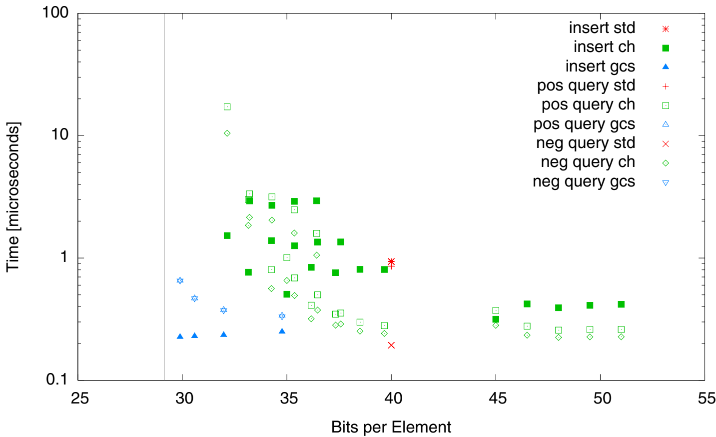


Fig. 5. Execution times of the different operations for many variants and tuning parameters, FPR equivalent to a standard Bloom filter with $c = 40$ and optimal k

Regarding the two-blocking variants, for positives and insertions, pat[1, 2] performs best, being up to almost twice as fast as any standard variant, for a low FPRs. It this range, pat[1, 2] can also compete for negative queries. When an even smaller FPR is requested, blo[2] should be used, which is only marginally slower than pat[1, 2].

We performed a similar test for the space-efficient replacements. The ch data structure used 3 additional bits per entry, while varying the load factor from 0.90 to 0.99.

The results are stated in Figure 5, comparing to the standard Bloom filter for $c = 40$, all featuring the same FPR. For this FPR, the lower bound in space for storing the hash values is $-\log 4.5110^{-9} + \log e = 29.14$ bits per element. The minimum space requirement in this experiment for gcs is in fact 29.14 bits ($I \to \infty$), reaching the optimum, while for ch, it is 30.80 bits ($\alpha \to 1$ and omitting one redundant helper bit per entry). For gcs, the index data structure can be easily and flexibly rebuilt after compact transmission, but for ch, the whole filter must be rebuilt to achieve acceptable execution times.

As we can see, the static gcs implementation provides excellent performance when the memory limitations are tight. If more space is available, Compact hash (ch) gives better query times, but collapses in terms of insertion performance.

7 Conclusion

Which variant or replacement of a Bloom filter works best depends on the application a lot. Standard Bloom filters are still a good choice in many cases. They are particularly efficient for negative queries. Even insertions and positive queries work better than one might think because modern hardware can mitigate cache faults by overlapping multiple memory accesses and because a reduction of k below the "optimal" value brings considerable speedup at moderate increase in space consumption or FPR. Blocked Bloom filters, possibly together with precomputed bit patterns, can mean a significant speedup if insertions and positive queries are important operations or when hash bits are expensive. Multiplexing and multiblocking Bloom filters become important when a very low FPR is required. Space-efficient Bloom filter replacements are particularly interesting when one wants to reduce communication volume for transferring the filter. Somewhat surprisingly, the price one pays in terms of access time is small or even negative if one uses our implementation based on bucketed Golomb coding. If internal space efficiency is less important than access time and saving communication volume, one could accelerate our implementation further by using Golomb coding only for the communication and by using a faster representation internally.

We believe that, independent of the particular results, our paper is also instructive as a case study in algorithm engineering and its methodology: Modeling both the machine (cache, prefetching, SIMD instructions) and the application (operation mix, difficulty of hashing) are very important here. The paper contains nontrivial components with respect to design, analysis, implementation,

experimental evaluation, and algorithm library design. In particular, the analysis is of a "nonstandard" type, i. e. , analysis only seems tractable with simplifying assumptions that are then validated experimentally.

Acknowledgments. We would like to thank M. Dietzfelbinger for valuable discussions and hints how to analyze the false positive rate of blocked Bloom filters. Frederik Transier provided a first implementation of Golomb coding.

References

1. Bloom, B.H.: Space-time trade-offs in hash coding with allowable errors. Communications of the ACM, vol. 13(7) (1970)
2. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics, vol. 1(4) (2004)
3. Cleary, J.G.: Compact hash tables using bidirectional linear probing. IEEE Transactions on Computers 33(9), 828–834 (1984)
4. Dillinger, P.C., Manolios, P.: Bloom filters in probabilistic verification. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 367–381. Springer, Heidelberg (2004)
5. Dillinger, P.C., Manolios, P.: Fast and accurate bitstate verification for SPIN. In: Graf, S., Mounier, L. (eds.) Model Checking Software. LNCS, vol. 2989, pp. 57–75. Springer, Heidelberg (2004)
6. Fan, L., Cao, P., Almeida, J.M., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM TON 8(3), 281–293 (2000)
7. Kirsch, A., Mitzenmacher, M.: Less hashing, same performance: Building a better Bloom filter. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 456–467. Springer, Heidelberg (2006)
8. Manber, U., Wu, S.: An algorithm for approximate membership checking with application to password security. Information Processing Letters 50(4), 191–197 (1994)
9. Mitzenmacher, M.: Compressed Bloom filters. In: PODC 2001, pp. 144–150 (2001)
10. Moffat, A., Turpin, A.: Compression and Coding Algorithms. Kluwer Academic Publishers, Dordrecht (2002)
11. Pagh, A., Pagh, R., Rao, S.S.: An optimal Bloom filter replacement. In: SODA 2005, pp. 823–829 (2005)
12. Sanders, P., Transier, F.: Intersection in integer inverted indices. In: ALENEX 2007 (2007)

Appendix

A Cache-Efficiency Evaluation

Table 3 lists the number of cache faults that are triggered by executing the experiment. We used a table of size either 64 KB, 2048 KB or 4096 KB. For $c = 20$, we inserted and queried 40,000,000 elements, respectively. blo[1] causes about one cache miss per operation, which is quite accurately reflected by the

Table 3. Number of cache misses for various algorithms, operations, and table sizes

	Table Size / Operation					
	64 KB		2048 KB		4096 KB	
Algorithm	insert / pos	neg	insert / pos	neg	insert / pos	neg
std	538601538	111606798	538864877	111633776	538878911	111644012
blo[1]	38564348	38547086	38539297	38522083	38508814	38489184
blo[2]	77068492	38929639	77079012	38936222	77077903	38937805
pat[1,1]	38616001	38567585	54681670	54577097	64018108	63925035
pat[1,2]	77270265	40841892	109155069	55279812	128127287	64866149
pat[2,1]	38656675	38602126	54665523	54575523	64065832	63974708
pat[2,2]	77278236	40868585	109409646	55392837	128379947	65000473
pat[3,1]	38657692	38606045	54601020	54510319	63965985	63891726
pat[3,2]	77203292	40808134	109413361	55396675	128109749	64862561

numbers. For a insertion or positively answered query, the number of cache faults is reduced by a factor of 13.96 compared to std. This is also just as expected, since $k = 14$. However, Figure 3 indicates that for $c = 20$, blo[1] is only about 3 times faster than std. Part of the explanation is that the number of hash bits needed by the two schemes only differs by a factor of about three. However, since the execution time is still dominated by the memory access times, an important part of the explanation seems to be that the compiler schedules memory accesses (or prefetches) already in the loop iteration before the actual use of the data. Thus, cache latency can be hidden behind other operations. This prefetching behavior also explains why there are about 2.9 cache faults per negative query of std although the analysis predicts only two. Apparently, once the query algorithm found a zero bit, one more memory access has already been issued.

For the two-blocking variants, the number of cache misses obviously doubles. When using patterns, the pattern table and the accessed blocks fight for the cache. When the table is as large as the cache, the numbers go up by a factor of 1.7, compared to a table of negligible size. But still, the number of cache misses is far lower than for the standard variants.

Crossing Minimization in Weighted Bipartite Graphs*

Olca A. Çakıroḡlu, Cesim Erten, Ömer Karataş, and Melih Sözdinler

Computer Science and Engineering, Işık University
Şile, Istanbul 34980, Turkey
arda.cakiroglu,cesim,omer.karatas,melih.sozdinler@isikun.edu.tr

Abstract. Given a bipartite graph $G = (L_0, L_1, E)$ and a fixed ordering of the nodes in L_0, the problem of finding an ordering of the nodes in L_1 that minimizes the number of crossings has received much attention in literature. The problem is NP-complete in general and several practically efficient heuristics and polynomial-time algorithms with a constant approximation ratio have been suggested. We generalize the problem and consider the version where the edges have nonnegative weights. Although this problem is more general and finds specific applications in automatic graph layout problems similar to those of the unweighted case, it has not received as much attention. We provide a new technique that efficiently approximates a solution to this more general problem within a constant approximation ratio of 3. In addition we provide appropriate generalizations of some common heuristics usually employed for the unweighted case and compare their performances.

1 Introduction

Given a bipartite graph $G = (L_0, L_1, E)$, the crossing minimization problem consists of finding an ordering of the nodes in L_0 and L_1 such that placing the two layers on two horizontal lines and drawing each edge as a straight line segment, the number of pairwise edge crossings is minimized. A related version is one where the ordering in one of the layers is already fixed. The former is usually referred to as the *both layers free bipartite crossing minimization* whereas the latter as the *one layer free bipartite crossing minimization*. Both problems have been extensively studied in literature. Unfortunately they are both NP-hard [6,10]. As a result extensive research has been devoted to the design of heuristics and approximation algorithms for these problems.

Both crossing minimization problems, especially the one layer free version, have been used as basic building blocks for automatic layout of directed graphs following the approach of Sugiyama, Tagawa, and Toda [18]. This approach consists mainly of three steps: Assigning the nodes to horizontal layers, ordering the nodes within each layer so as to minimize the number of crossings, and

* Partially supported by TUBITAK-The Scientific and Technological Research Council of Turkey, grant 106E071.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 122–135, 2007.
© Springer-Verlag Berlin Heidelberg 2007

finally assigning actual coordinates for the nodes and edge bends. A commonly used approach for the second step is the "layer-by-layer sweep" method which requires a solution to the one layer free crossing minimization problem.

The current paper examines the weighted generalization of the (unweighted) one layer free crossing minimization problem, from now on referred to as OLF. Specifically we consider the following:

Weighted One Layer Free Problem (WOLF): Given an edge-weighted bipartite graph $G = (L_0, L_1, E)$ and a fixed ordering of nodes in L_0, find an ordering of nodes in L_1 such that the *total weighted crossings* in the resulting drawing is minimized. If two edges $e_1, e_2 \in E$ cross then this crossing amounts to $\mathcal{W}(e_1) \times \mathcal{W}(e_2)$ in the total weighted crossings, where $\mathcal{W}(e_1), \mathcal{W}(e_2)$ denote the weights of e_1, e_2 respectively. We assume all weights are nonnegative.

Besides the fact that it is a generalization of OLF, its wide range of applications provides further motivation to study WOLF. Similar to OLF, natural applications include those related to computing layered layouts of directed graphs. Many such instances assign an edge weight to indicate its "importance". The goal then is to compute layouts with few crossings between important edges of large weight [3,4,8]. Other applications specific to WOLF include recent problems related to wire crossing minimization in VLSI [16].

1.1 Previous Work

Crossing minimization problems in drawings of bipartite graphs have been studied extensively in literature. A common method to solve the both layers free version of the problem is to iteratively apply a solution to OLF, while alternating the fixed layer at each iteration. Therefore considerable attention has been devoted to the OLF problem and its variations [2,5,6,7,11,12,14,15,17,19].

Since OLF is NP-complete even when the graph instances are sparse [14], much of related research concentrates on the design of heuristics and approximation algorithms. Most popular heuristics include the *barycenter* method of [18] and the *median heuristic* of [6]. Jünger and Mutzel survey various heuristics and experimentally compare their performances [11]. They conclude that the barycenter method yields slightly better results than the median heuristic in practice. On the other hand from a theoretical point of view median heuristic is better. Specifically, they both run in linear time and the median heuristic is a 3-approximation, whereas the approximation ratio of the barycenter method is $\Theta(\sqrt{|L_0|})$ [6]. Yamaguchi and Sugimoto [19] provide a greedy algorithm GRE that has the same approximation ratio of 3 in the worst case and that works well in practice. However the running time of GRE is quadratic. Recently, Nagamochi devised a 1.47-approximation algorithm for OLF [15].

Another promising technique for OLF is the penalty graph approach introduced by Sugiyama *et al.* [18]. The performance of this method depends on an affective solution to the minimum feedback arc set (FAS) problem which is also NP-complete [9]. Demetrescu and Finocchi experimentally compare the performance of the penalty graph method based on their algorithm for FAS to that of the barycenter, median, and the GRE heuristics [5].

Applications requiring a solution to WOLF usually employ a weighted modification of the barycenter method [3,4,8] or a penalty graph based approach [16]. Such practices are based on the presupposition that simple extensions of desirable methods for OLF should also lead to efficient solutions for WOLF. To the contrary, our experiments indicate this may not necessarily be the case.

1.2 Our Contributions

To the best of our knowledge this is the first study to consider specifically the WOLF problem and to compare various promising methods. The following summarizes the main contributions:

- We provide an efficient approximation algorithm `3-WOLF` for the WOLF problem. Specifically, algorithm `3-WOLF` 3-approximates WOLF in $O(|E| + |V| \log |V|)$ time, where $V = L_0 \cup L_1$. We note that this is the first polynomial time algorithm to have a constant approximation ratio for the WOLF problem. Although there are several polynomial-time, constant approximations for OLF [6,15,19], it is not obvious how to generalize them for WOLF while retaining the same approximation ratio.
- We devise weighted modifications of common heuristics that are shown to have a good performance for OLF in practical settings. Specifically we present extensions of the barycenter, median, GRE, and the penalty graph methods previously suggested for OLF.
- We experimentally compare the performances of these methods to that of `3-WOLF`. Our experiments indicate that the performance of `3-WOLF`, in terms of the resulting number of crossings, is better than the performances of the methods with comparable running times. Besides, its performance is almost the same as that of the methods with expensive running time requirements.
- The distinction between efficient methods for the OLF and WOLF problems arises as an interesting result of our experiments. A particular instance of this is the barycenter method. Although it seems to outperform the methods with comparable running times in the OLF settings, its weighted extension does not share the same performance results in the WOLF settings.

2 `3-WOLF`: A 3-Approximation Algorithm for WOLF

Given a bipartite graph $G = (L_0, L_1, E)$, let n_0, n_1 denote the sizes of the layers L_0, L_1 respectively. Assume the nodes in L_0 are labeled with integers from 1 through n_0. For $u \in L_1$ and $j \leq l$, let $\mathcal{W}(u)_j^l = \sum_{p=j}^{l} \mathcal{W}(u,p)$, where $\mathcal{W}(u,p)$ is the weight of the edge (u,p) for $p \in L_0$. With this notation we assume that $\mathcal{W}(u,p) = 0$ if there is no edge between u and p. We also note that all weights are assumed to be nonnegative. For $u, v \in L_1$, let c_{uv} denote the sum of the weighted crossings between the edges incident on u and the edges incident on v when u is placed to the left of v, that is $c_{uv} = \sum_{p=1}^{n_0-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^{n_0}$.

Algorithm `3-WOLF` consists mainly of two phases: A coarse-grained ordering phase followed by a fine-grained ordering phase. The initial coarse-grained phase

```
Algorithm 1: Coarse-grained Ordering
```

/*Initially each partition $\mathcal{P}_r$, where $0 \le r \le n_0 - 1$, is empty.*/
forall $u \in L_1$ **do**
  $leftsum = 0$; $rightsum = \mathcal{W}(u)_2^{n_0}$;
  for $r \leftarrow 0$ **to** $n_0 - 1$ **do**
    if $leftsum \ge rightsum$ **then** break;
    $leftsum = leftsum + \mathcal{W}(u, r+1)$;
    $rightsum = rightsum - \mathcal{W}(u, r+2)$;
  $\mathcal{P}_r = \mathcal{P}_r \cup \{u\}$;
/*Partitions are ordered according to indices: $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n_0-1}$*/

partitions L_1 into disjoint sets and orders the partitions. Then the second phase orders the nodes within each partition independently (without considering the the nodes in other partitions).

2.1 Phase-1: Coarse-Grained Ordering

To make the description easier, for now we assume that G is a complete weighted bipartite graph (some weights can be zero). We later provide the details necessary to implement it more efficiently.

We partition L_1 into n_0 disjoint sets $\mathcal{P}_0, \ldots \mathcal{P}_{n_0-1}$ where $L_1 = \bigcup_{0 \le r \le n_0-1} \mathcal{P}_r$. For ease of notation let $\mathcal{W}(u)_1^0 = \mathcal{W}(u)_{n_0+1}^{n_0} = 0$, for $u \in L_1$. We define initial partition $\mathcal{P}_0$ as, $\mathcal{P}_0 = \{u \in L_1 | \mathcal{W}(u)_1^0 \ge \mathcal{W}(u)_2^{n_0}\}$. For $r \ge 1$, $\mathcal{P}_r$ is defined as:

$$\mathcal{P}_r = \{u \in L_1 | \mathcal{W}(u)_1^{r-1} < \mathcal{W}(u)_{r+1}^{n_0} \text{ and } \mathcal{W}(u)_1^r \ge \mathcal{W}(u)_{r+2}^{n_0}\}$$

Obviously each node in L_1 belongs to exactly one partition. The partitions are ordered in the increasing order of their indices from left to right. Algorithm 1 provides a pseudocode for Phase-1. The following lemma shows the correctness of the described partitioning and ordering:

Lemma 1. *Given $u \in \mathcal{P}_q$ and $v \in \mathcal{P}_r$, where $q < r$, after Phase-1 of Algorithm* 3-WOLF, *$c_{uv} \le 3c_{vu}$.*

Proof. If $q = 0$ then $\mathcal{W}(u)_2^{n_0} = 0$. Since $q < r$ we have $\mathcal{W}(v)_2^{n_0} > 0$. This implies $c_{uv} \le c_{vu}$ and the lemma holds trivially. Now assume $q > 0$. Since $u \in \mathcal{P}_q$ and $v \in \mathcal{P}_r$, by definition the following inequalities hold:

$$\mathcal{W}(u)_1^{q-1} < \mathcal{W}(u)_{q+1}^{n_0} \text{ and } \mathcal{W}(u)_1^q \ge \mathcal{W}(u)_{q+2}^{n_0} \quad (1)$$

$$\mathcal{W}(v)_1^{r-1} < \mathcal{W}(v)_{r+1}^{n_0} \text{ and } \mathcal{W}(v)_1^r \ge \mathcal{W}(v)_{r+2}^{n_0} \quad (2)$$

We write $c_{uv} = A + B + C$ and $c_{vu} = A' + B' + C'$ where,

$$A = \mathcal{W}(v)_1^q \times \mathcal{W}(u)_{q+1}^{n_0},\ B = \sum_{p=1}^{q-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^q,\ C = \sum_{p=q+1}^{n_0-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^{n_0}$$

$$A' = \mathcal{W}(u)_1^q \times \mathcal{W}(v)_{q+1}^{n_0},\ B' = \sum_{p=1}^{q-1} \mathcal{W}(u,p) \times \mathcal{W}(v)_{p+1}^q,\ C' = \sum_{p=q+1}^{n_0-1} \mathcal{W}(u,p) \times \mathcal{W}(v)_{p+1}^{n_0}$$

We show that $A, B, C \leq c_{vu}$. We have $\mathcal{W}(u)_1^q \geq \mathcal{W}(u)_{q+2}^{n_0}$ by (1). This implies $\mathcal{W}(u)_1^{q+1} \geq \mathcal{W}(u)_{q+1}^{n_0}$. Since $q < r$, we have $\mathcal{W}(v)_1^q < \mathcal{W}(v)_{r+1}^{n_0}$ by the first inequality in (2). Putting together we get,

$$\mathcal{W}(v)_1^q \times \mathcal{W}(u)_{q+1}^{n_0} \leq \mathcal{W}(u)_1^{q+1} \times \mathcal{W}(v)_{r+1}^{n_0}$$

Since $q < r$, the right side is at most $\mathcal{W}(u)_1^q \times \mathcal{W}(v)_{q+1}^{n_0} + \mathcal{W}(u, q+1) \times \mathcal{W}(v)_{r+1}^{n_0}$. The first term in this sum is equal to A'. The second term is at most C'. Therefore it follows that $A \leq c_{vu}$.

To prove it for B, we have $\sum_{p=1}^{q-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^q \leq \mathcal{W}(u)_2^q \times \mathcal{W}(v)_1^{q-1}$. Since $q < r$, by (2) we have that $\mathcal{W}(v)_1^{q-1} < \mathcal{W}(v)_{r+1}^{n_0}$, which further implies $\mathcal{W}(v)_1^{q-1} < \mathcal{W}(v)_{q+1}^{n_0}$. Putting together, we have $B \leq A'$ and therefore $B \leq c_{vu}$. Similarly for C, we get $\sum_{p=q+1}^{n_0-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^{n_0} \leq \mathcal{W}(u)_{q+2}^{n_0} \times \mathcal{W}(v)_{q+1}^{n_0-1}$. We have $\mathcal{W}(u)_1^q \geq \mathcal{W}(u)_{q+2}^{n_0}$ by (1) which implies $C \leq A'$ and therefore $C \leq c_{vu}$.

2.2 Phase-2: Fine-Grained Ordering

Let $\pi(\mathcal{P}_r)$ be a permutation of the nodes in $\mathcal{P}_r$. We define the following invariant:

Definition 1. *Given a partition $\mathcal{P}_r$ and S such that $S \subseteq \mathcal{P}_r$, let $\pi(S)$ be a permutation of S. We call $\pi(S)$ a* **partition invariant satisfying permutation (PISP)** *if for any $u \in S$ and for all $v \in S \backslash \{u\}$ that is placed to the right of u in $\pi(S)$ the following holds:*

$$\mathcal{W}(v)_1^r \times \mathcal{W}(u)_{r+1}^{n_0} \leq \mathcal{W}(u)_1^r \times \mathcal{W}(v)_{r+1}^{n_0}$$

We show that an algorithm that orders $\mathcal{P}_r$ according to the partition invariant is appropriate for our purposes:

Lemma 2. *Let $\pi(\mathcal{P}_r)$ be a PISP of $\mathcal{P}_r$. For any pair $u, v \in \mathcal{P}_r$ where u is to the left of v in $\pi(\mathcal{P}_r)$ we have $c_{uv} \leq 3c_{vu}$.*

Proof. We write $c_{uv} = A + B + C$ and $c_{vu} = A' + B' + C'$ where,

$$A = \mathcal{W}(v)_1^r \times \mathcal{W}(u)_{r+1}^{n_0},\ B = \sum_{p=1}^{r-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^r,\ C = \sum_{p=r+1}^{n_0-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^{n_0}$$

$$A' = \mathcal{W}(u)_1^r \times \mathcal{W}(v)_{r+1}^{n_0},\ B' = \sum_{p=1}^{r-1} \mathcal{W}(u,p) \times \mathcal{W}(v)_{p+1}^r,\ C' = \sum_{p=r+1}^{n_0-1} \mathcal{W}(u,p) \times \mathcal{W}(v)_{p+1}^{n_0}$$

We show that $A, B, C \leq A'$. This is true for A since $\pi(\mathcal{P}_r)$ is a PISP.

For B, we note that $\sum_{p=1}^{r-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^{r} \leq \mathcal{W}(u)_1^r \times \mathcal{W}(v)_1^{r-1}$. From the definition of $\mathcal{P}_r$ we have $\mathcal{W}(v)_1^{r-1} < \mathcal{W}(v)_{r+1}^{n_0}$. Therefore the inequality $\sum_{p=1}^{r-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^{r} \leq \mathcal{W}(u)_1^r \times \mathcal{W}(v)_{r+1}^{n_0}$ holds, which implies $B \leq A'$.

For C, we note that $\sum_{p=r+1}^{n_0-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^{n_0} \leq \mathcal{W}(u)_{r+2}^{n_0} \times \mathcal{W}(v)_{r+1}^{n_0-1}$. From the definition of $\mathcal{P}_r$ we have $\mathcal{W}(u)_1^r \geq \mathcal{W}(u)_{r+2}^{n_0}$. Therefore the inequality $\sum_{p=r+1}^{n_0-1} \mathcal{W}(v,p) \times \mathcal{W}(u)_{p+1}^{n_0} \leq \mathcal{W}(u)_1^r \times \mathcal{W}(v)_{r+1}^{n_0}$ holds and $C \leq A'$.

We show that we can construct a PISP efficiently. The following transitivity lemma will be helpful for further results.

Lemma 3. *Given $u, v, w \in \mathcal{P}_r$, each with degree at least one, assume the following inequalities hold for some j, $1 \leq j \leq n_0$:*

$$\mathcal{W}(v)_1^j \times \mathcal{W}(u)_{j+1}^{n_0} \leq \mathcal{W}(u)_1^j \times \mathcal{W}(v)_{j+1}^{n_0} \tag{3}$$

$$\mathcal{W}(w)_1^j \times \mathcal{W}(v)_{j+1}^{n_0} \leq \mathcal{W}(v)_1^j \times \mathcal{W}(w)_{j+1}^{n_0} \tag{4}$$

Then $\mathcal{W}(w)_1^j \times \mathcal{W}(u)_{j+1}^{n_0} \leq \mathcal{W}(u)_1^j \times \mathcal{W}(w)_{j+1}^{n_0}$.

Proof. Multiplying both sides of the inequality (3) with $\mathcal{W}(w)_1^j$ and replacing $\mathcal{W}(v)_{j+1}^{n_0} \times \mathcal{W}(w)_1^j$ with $\mathcal{W}(v)_1^j \times \mathcal{W}(w)_{j+1}^{n_0}$ we get:

$$\mathcal{W}(v)_1^j \times \mathcal{W}(u)_{j+1}^{n_0} \times \mathcal{W}(w)_1^j \leq \mathcal{W}(u)_1^j \times \mathcal{W}(v)_1^j \times \mathcal{W}(w)_{j+1}^{n_0} \tag{5}$$

If $\mathcal{W}(v)_1^j \neq 0$, we can divide both sides of (5) with $\mathcal{W}(v)_1^j$ and the lemma holds. On the other hand if $\mathcal{W}(v)_1^j = 0$, then $\mathcal{W}(v)_{j+1}^{n_0} \neq 0$ since v has degree at least one. This implies $\mathcal{W}(w)_1^j = 0$ and the lemma holds trivially.

Constructing $\boldsymbol{\pi(\mathcal{P}_r)}$ from $\boldsymbol{\mathcal{P}_r}$: We assume all nodes in $\mathcal{P}_r$ have degree at least one, as nodes with degree zero can be placed arbitrarily. We use a divide-and-conquer approach. We first divide $\mathcal{P}_r$ into two subsets of equal size $\mathcal{P}_{r1}, \mathcal{P}_{r2}$. We solve the problem for those subsets to obtain two PISPs for $\mathcal{P}_{r1}, \mathcal{P}_{r2}$ and finally we merge the resulting PISPs. The details are presented in Algorithm 2.

Assuming $\pi(\mathcal{P}_{r1}), \pi(\mathcal{P}_{r2})$ are PISPs, we need to prove the correctness of the merge procedure. We do so inductively. Let $\pi(\mathcal{P}_r)^t$ denote $\pi(\mathcal{P}_r)$ after t steps of the merge procedure. Let u', v' be the current nodes of $\pi(\mathcal{P}_{r1}), \pi(\mathcal{P}_{r2})$ respectively at the end of t steps. Let '$\circ$' denote the concatenation operator.

Lemma 4. *$\pi(\mathcal{P}_r)^t \circ \{u'\}$ and $\pi(\mathcal{P}_r)^t \circ \{v'\}$ are PISPs, where $0 \leq t \leq |\mathcal{P}_r|$.*

Proof. The proof is by induction on t. The base case of $t = 0$ holds trivially. Let u, v be the current nodes in $\pi(\mathcal{P}_{r1}), \pi(\mathcal{P}_{r2})$ respectively, at the beginning of step t. Assume inductively that $\pi(\mathcal{P}_r)^{t-1} \circ \{u\}$ and $\pi(\mathcal{P}_r)^{t-1} \circ \{v\}$ are PISPs.

Without loss of generality assume

$$\mathcal{W}(v)_1^r \times \mathcal{W}(u)_{r+1}^{n_0} \leq \mathcal{W}(u)_1^r \times \mathcal{W}(v)_{r+1}^{n_0} \tag{6}$$

Algorithm 2: Fine-grained Ordering

```
/*Nodes in P_r have nonzero degrees. Initially π(P_r) is empty. */
if |P_r| = 1 then
    π(P_r) = {u} where u ∈ P_r;
else
    Divide P_r into partitions of equal size P_r1, P_r2;
    /*Solve for each partition.*/
    π(P_r1) = Fine-grained Ordering P_r1;
    π(P_r2) = Fine-grained Ordering P_r2;
    /*Merge the resulting permutations.*/
    New node a s.t. W(a)_1^r = −1, W(a)_{r+1}^{n_0} = 0;
    π(P_r1) = π(P_r1) ∘ {a};
    π(P_r2) = π(P_r2) ∘ {a};
    Let u, v be the first nodes in π(P_r1), π(P_r2) respectively;
    for t ← 1 to |P_r| do
        if W(v)_1^r × W(u)_{r+1}^{n_0} ≤ W(u)_1^r × W(v)_{r+1}^{n_0} then
            π(P_r) = π(P_r) ∘ {u};
            u = u' where u' follows u in π(P_r1);
        else
            π(P_r) = π(P_r) ∘ {v};
            v = v' where v' follows v in π(P_r2);
```

Note that this implies $v' = v$. We show the following:

1. $\pi(\mathcal{P}_r)^{t-1} \circ \{u\} \circ \{u'\}$ is a PISP:
   We note that
$$\mathcal{W}(u')_1^r \times \mathcal{W}(u)_{r+1}^{n_0} \leq \mathcal{W}(u)_1^r \times \mathcal{W}(u')_{r+1}^{n_0} \tag{7}$$
   as u is to the left of u' in $\pi(\mathcal{P}_{r1})$ which is a PISP.
   We need to show that $\mathcal{W}(u')_1^r \times \mathcal{W}(x)_{r+1}^{n_0} \leq \mathcal{W}(x)_1^r \times \mathcal{W}(u')_{r+1}^{n_0}$ for all $x \in \pi(\mathcal{P}_r)^{t-1} \circ \{u\}$. If $x = u$ the inequality is satisfied because of (7). On the other hand if $x \neq u$, we have $\mathcal{W}(u)_1^r \times \mathcal{W}(x)_{r+1}^{n_0} \leq \mathcal{W}(x)_1^r \times \mathcal{W}(u)_{r+1}^{n_0}$ since by the inductive hypothesis $\pi(\mathcal{P}_r)^{t-1} \circ \{u\}$ is a PISP. Using the transitivity property stated in Lemma 3, we combine this last inequality and that of (7) which implies $\mathcal{W}(u')_1^r \times \mathcal{W}(x)_{r+1}^{n_0} \leq \mathcal{W}(x)_1^r \times \mathcal{W}(u')_{r+1}^{n_0}$.
2. $\pi(\mathcal{P}_r)^{t-1} \circ \{u\} \circ \{v\}$ is a PISP:
   We need to show that $\mathcal{W}(v)_1^r \times \mathcal{W}(x)_{r+1}^{n_0} \leq \mathcal{W}(x)_1^r \times \mathcal{W}(v)_{r+1}^{n_0}$ for all $x \in \pi(\mathcal{P}_r)^{t-1} \circ \{u\}$. If $x = u$ the inequality is satisfied because of the initial assumption in (6). On the other hand if $x \neq u$ the inequality holds by the inductive hypothesis.

The lemma below is an immediate consequence of Lemma 4:

Lemma 5. *Let $\pi(\mathcal{P}_r)$ be the output permutation of Algorithm 2 applied on $\mathcal{P}_r$. $\pi(\mathcal{P}_r)$ is a PISP.*

The correctness of Phase-2 follows from Lemma 2 and Lemma 5:

Lemma 6. *Given $u, v \in \mathcal{P}_r$, where u is placed to the left of v after Phase-2 of Algorithm* `3-WOLF`, *$c_{uv} \leq 3c_{vu}$.*

Running Time of Algorithm 3-WOLF: In order to implement Phase-1 efficiently we note that a node $u \in L_1$ may not be connected to all the nodes in L_0. Let $\mathcal{N}_u$ denote the set of neighbors of u. We assume the nodes in L_0 are labeled from 1 to $|L_0|$ and that nodes in $\mathcal{N}_u$ are already in sorted order according to these labels. We compute the sum $\sum_{i=1}^{|\mathcal{N}_u|} \mathcal{W}(u, \mathcal{N}_u[i])$ exactly once at the beginning. Here $\mathcal{N}_u[i]$ indicates the ith neighbor's label. Going through every neighbor of u we increment *leftsum* value each time, and decrement the *rightsum* value only whenever necessary, where leftsum indicates $\mathcal{W}(u)_1^k$ and rightsum indicates $\mathcal{W}(u)_{k+2}^{n_0}$, for $0 \leq k < |\mathcal{N}_u|$. The partition of u is found once the leftsum value is greater than or equal to the rightsum value. We defer the rest of the implementation details in the form of a pseudo-code to the full version of the paper. The running time required by this implementation of Phase-1 is $O(|E| + |L_0| + |L_1|)$. We assume the values $\mathcal{W}(u)_1^r, \mathcal{W}(u)_{r+1}^{n_0}$ are also recorded during Phase-1, once they are computed. Therefore following the description of Phase-2 in Algorithm 2, the second phase requires time $O(|L_0| + |L_1| \log |L_1|)$. The theorem below summarizes the main result:

Theorem 1. *Given a bipartite graph $G = (L_0, L_1, E)$, algorithm* 3-WOLF *3-approximates WOLF in time $O(|E| + |L_0| + |L_1| \log |L_1|)$.*

Proof. For a given input graph $G = (L_0, L_1, E)$, a trivial lowerbound on the number of crossings is $LB = \sum_{u,v \in L_1} min(c_{uv}, c_{vu})$. The proof then follows the correctness results presented in Lemma 1 and Lemma 6, and the discussion above regarding the running time.

Remark 1. We show that there exist instances of WOLF for which the bounds of Lemma 1 and Lemma 6 are almost met. Therefore the performance ratio analysis of those lemmas are tight in the worst case. We defer the details regarding the constructions of such instances to the full version.

3 Weighted Modifications of Common Methods for OLF

In addition to 3-WOLF, we consider modifications of four well-known methods suggested previously for OLF: The median, barycenter, GRE heuristics, and the penalty graph method.

W-MED **(Weighted Median):** The original median algorithm as described by Eades and Wormald [6], assigns x-coordinate of $u \in L_1$ to be the median of the x-coordinates of the nodes in $\mathcal{N}_u$. If two nodes are assigned the same median, then one of them is placed to the left randomly, except when one has odd degree, and the other even, in which case the odd degree node is placed to the left.

The weighted version we propose, W-MED, in essence is similar to 3-WOLF. Algorithm W-MED also proceeds in two phases. In the coarse-grained phase we decide on the partition of each node $u \in L_1$. Node u is placed in $\mathcal{P}_r$, where r is the smallest integer value such that $\mathcal{W}(u)_1^r \geq \mathcal{W}(u)_{r+1}^{n_0}$. The partitions are then ordered from left to right in the increasing order of their indices. In the second phase of W-MED we apply W-BARY, described next, on each partition $\mathcal{P}_r$.

Remark 2. We note that the initial coarse-grained phase of W-MED is analogous to the median assignment in the OLF settings, assuming the medians are different. Consider the unweighted graph constructed by replacing each neighbor $\mathcal{N}_u[i]$ of $u \in L_1$ with $\mathcal{W}(u, \mathcal{N}_u[i])$ artificial nodes and connecting each one to u with an unweighted edge. Let x be the artificial node that is computed as the median of u as a result of applying the original median algorithm on the unweighted graph. Then the median node (the partition $\mathcal{P}_r$) picked by the first phase of W-MED is the one that is replaced by the artifical nodes including x. Since the original median algorithm provides a guaranteed constant approximation ratio of 3, it is natural to expect a good performance from the first phase of W-MED. In fact we can prove that nodes in different partitions are placed appropriately, up to the constant approximation ratio of 3, after this initial phase. Details of this result, analogous to Lemma 1 of 3-WOLF are deferred to the full version. For the second phase, in OLF settings, the median algorithm simply checks the node degrees to order the nodes within each partition to guarantee the same approximation ratio. However similar reasoning does not seem to apply to WOLF.

W-BARY **(Weighted Barycenter):** The original barycenter method assigns the x-coordinate of each $u \in L_1$ as the average of the x-coordinates of its neighbors [18]. In W-BARY edge weights are introduced to this average. That is, the x-coordinate of u is assigned to

$$\frac{\sum_{i=1}^{|\mathcal{N}_u|}(\mathcal{W}(u, \mathcal{N}_u[i]) \times \mathcal{N}_u[i])}{\sum_{i=1}^{|\mathcal{N}_u|}\mathcal{W}(u, \mathcal{N}_u[i])}$$

Remark 3. The approximation ratio of the barycenter method applied to OLF is $\Theta(\sqrt{|L_0|})$ [6]. In contrast, there exist instances of WOLF for which W-BARY produces outputs where the performance ratio is $\Omega(|L_0| + |L_1|)$. Interestingly such instances are plausible even when degrees are restricted to 2. We note that in OLF settings the barycenter method achieves an optimal solution under this restriction [14]. Details are deferred to the full version.

W-GRE **(Weighted GRE):** The GRE algorithm described for OLF greedily assigns $u \in L_1$ as the next node to place in the rightmost position [19]. It does so by choosing u that minimizes

$$\frac{\sum_{v \in L_1'} c_{uv}}{\sum_{v \in L_1'} min(c_{uv}, c_{vu})}$$

among all not yet placed nodes, denoted by L_1'. The W-GRE algorithm works the same way, except edge weights are taken into account while computing c_{uv}.

W-PM **(Weighted Penalty Minimization):** The PM algorithm for OLF starts with the construction of a weighted directed graph called the penalty graph. The node set is that of L_1. An edge with weight $c_{vu} - c_{uv}$ from u to v is inserted in the

penalty graph if $c_{uv} < c_{vu}$. The algorithm then seeks for a minimum feedback arc set (FAS) in the penalty graph [18]. Demetrescu and Finocchi propose an implementation of the PM method based on their algorithm for approximating FAS [5]. The `W-PM` method is based on their implementation, except as with `W-GRE`, the computation of c_{uv} takes into account the edge weights.

Remark 4. All algorithms assume $\mathcal{N}_u$ is sorted for $u \in L_1$. Therefore all running times presented exclude such a cost. Both `W-MED` and `W-BARY` run in linear time. The running time of `W-GRE` is $O(|E|^2 + |L_1|^2)$ and that of `W-PM` is $O(|E|^2 + |L_1|^4)$. Both `W-GRE` and `W-PM` require the computation of a cross table. All c_{uv} values are retrieved from this table which is computed beforehand. A straightforward implementation of this computation requires time $O(|E|^2)$. In OLF settings it can be implemented in time $O(|E| \times |L_1|)$. This improvement is based on the observation that each crossing increments the total crossings by the same amount of 1. However this is no longer true in WOLF settings. Therefore our construction of the cross table requires $O(|E^2|)$ time, although this does not have too much affect on the actual CPU times required by the algorithms.

4 Experimental Setup and Results

We implemented all the algorithms in C++ using the LEDA library [13]. The implementations are freely available in [1]. Experiments are performed on computers with the configuration of P4 3.2 GHz of CPU and 1GB of RAM.

4.1 Random Graph Generator and Parameters

We constructed a parameter list that is relevant to the WOLF problem. We implemented a random graph generator that takes as input an instance of such a list and produces an output graph compatible with the requirements of that instance. Each output graph is then fed to all the algorithms and the results are recorded. We repeat this sequence of calls to graph generator followed by calls to algorithms k times, where k is assigned a value in the range from 5 to 50 depending on the edge density. An average of the resulting crossing numbers from these k iterations are computed.

The implementation of the random graph generator is also available in [1]. We briefly describe the problem parameters. Two items included in our parameter list are n_0, n_1 which represent the sizes of the layers L_0, L_1 respectively. The rest of the items are as follows: *Edge Density* (ED) is the probability of creating an edge between any pair of nodes and varies between 0.001 and 1. *Average Weight* (AW) is the expected average weight of a randomly generated edge. *Weight Balance* (WB) is a parameter we introduce for the random assignment of weights and it is used to compute an upperbound on the weight of an edge. Specifically, it represents the ratio of sum of weights in the graph to the maximum potential weight to be assigned to an edge.

Random graph generator consists of two phases. The first phase creates a random unweighted graph that satisfies the ED constraint. To this aim, for

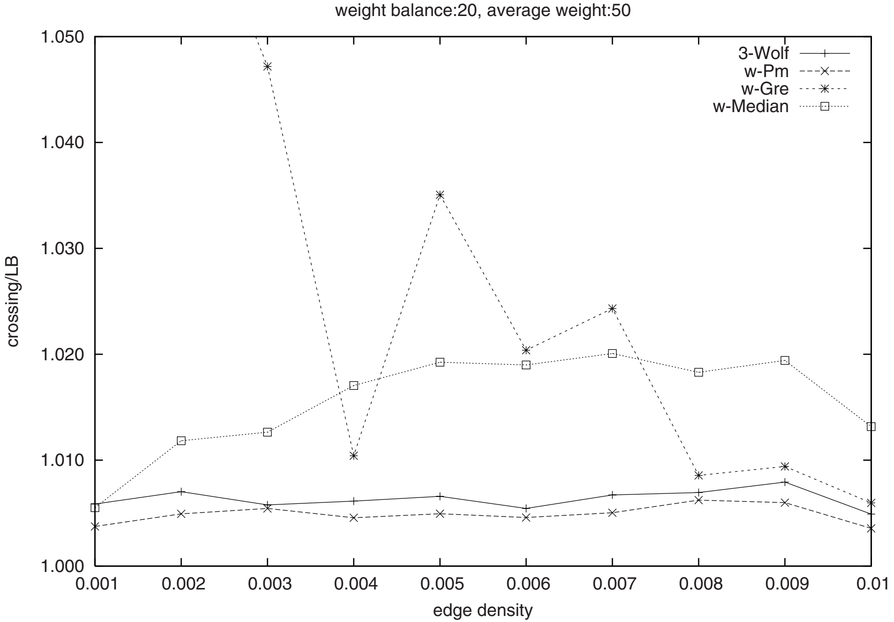


Fig. 1. Quality of the outputs, in terms of the *crossing*/*LB* values, provided by the algorithms for the sparsest graphs experimented (when edge density is in the range $[0.001, 0.01]$)

each pair of nodes (u, p), where $u \in L_1, p \in L_0$, a uniform random number $t \in [0, 1]$ is generated for every pair. If $t \leq ED$ then the edge (u, p) is added to E. After this phase the expected number of edges is $n_0 \times n_1 \times ED$. The second phase consists of assigning random weights to the set of edges in a way that AW constraint is satisfied. First a random permutation of E is created. Each edge $e \in E$ is selected in this order and $\mathcal{W}(e)$ is increased by a uniform random integer w where $0 \leq w < SW/WB$. Here SW indicates *Sum of Weights* which is equal to $n_0 \times n_1 \times ED \times AW$ initially. Its value is decreased by w after each weight assignment to an edge e. Intuitively SW/WB indicates a continually decreasing upper bound for the randomly generated weight.

4.2 Experiments and Discussion of Results

We have two performance measures. We compute *crossing*/*LB* values for each algorithm to measure the quality of its solution for WOLF. Here *crossing* indicates the total weighted crossings that arise in the output layout of an algorithm. The second measure is the running time required by each algorithm. All experiments are performed on randomly generated weighted bipartite graphs with 500 nodes in both layers L_0 and L_1.

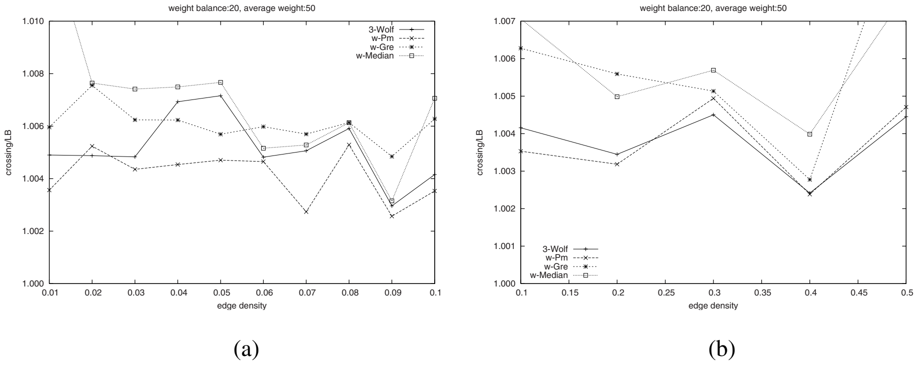


Fig. 2. (a) Quality of the outputs provided by the algorithms when edge density is in the range $[0.01, 0.1]$; **(b)** Quality of the outputs provided by the algorithms for the densest graphs experimented (when edge density is in the range $[0.1, 0.5]$)

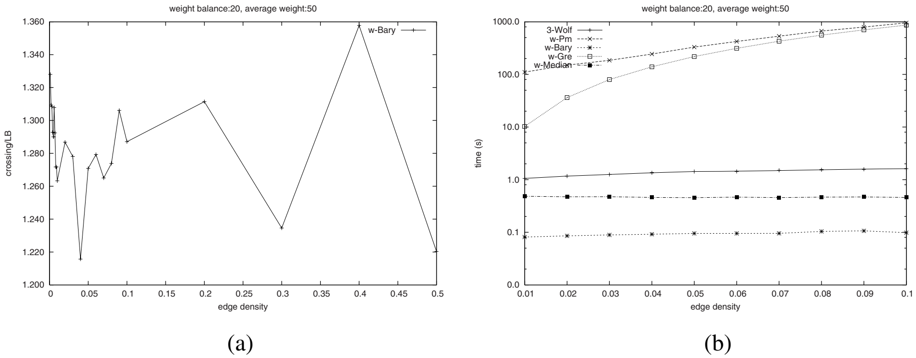


Fig. 3. (a) Quality of the outputs provided by W-BARY for the complete range $[0.001, 0.01]$); **(b)** Running times measured in terms of the actual CPU times required

The quality of results in terms of $crossing/LB$ values are depicted in Figures 1, 2-a and 2-b, where edge density varies in the ranges $[0.001, 0.01]$, $[0.01, 0.1]$, and $[0.1, 0.5]$ respectively. These figures include the results of all algorithms except for W-BARY which provided very large $crossing/LB$ values. Therefore, to provide a better visualization of the comparison between the rest of the algorithms, the results of W-BARY for the complete density range of $[0.001, 0.5]$ are depicted separately in Figure 3-a. We can summarize the quality results for almost all the ranges as follows: W-PM had the best results. 3-WOLF performed almost as good as W-PM (sometimes even better). W-GRE and W-MED followed them with similar performances. No clear winner among the two could be decided. Finally W-BARY provided the worst results in all the ranges and was clearly separated from the rest of the algorithms with its poor performance.

Figure 3-b depicts the running time results in terms of the measured CPU time required by each algorithm. We provide this measure only for the range $[0.01, 0.1]$

since the plots for the rest of the ranges were almost the same. We note that the running time plots are log-scale, therefore the actual time differences between the algorithms W-GRE, W-PM and the algorithms 3-WOLF, W-MED, W-BARY are much larger than visualized in the plots.

It is interesting to note that even though the time requirements of W-GRE and W-PM is much larger than that of 3-WOLF (almost 10000 times as much for dense graphs) the output qualities measured in terms of $crossing/LB$ values are almost the same. Another interesting result of our experiments is the distinction between the settings of OLF and WOLF. In the OLF settings the barycenter method has remarkable performance given its running time requirement. The median algorithm seems not to perform well in practice in this case. Additionally, the quality of results produced by the GRE and the PM methods is so good that they may be appealing even with such poor performance in terms of running time. However in the WOLF settings W-BARY is the worst in terms of the crossings produced. The intuition behind this bad performance in terms of the quality of results is in part provided in Remark 3. Furthermore the quality of results produced by the W-GRE, W-PM algorithms is not remarkable anymore. In fact even a simple combination of the median and the barycenter heuristics, W-MED, provides similar quality.

5 Conclusion

The extensively studied OLF problem is that of minimizing edge crossings in bipartite graphs when one layer is fixed. We generalized the problem to WOLF, where the edges of the bipartite graph have nonnegative weights. We presented an algorithm, 3-WOLF, that provides a 3-approximation for the WOLF problem. Although several algorithms with constant approximation ratios for OLF were known previously, this is the first analogous result regarding WOLF. We also provided appropriate generalizations of methods commonly suggested for OLF. We experimentally evaluated all the suggested algorithms. Our experiments indicate that 3-WOLF not only has the advantage of a theoretically proven constant approximation ratio, it also has good perfomance in terms of the quality of the output and the time required to compute it.

References

1. Wolf implementations. www2.isikun.edu.tr/personel/cesimerten/wolf.rar
2. Biedl, T., Brandenburg, F.J., Deng, X.: Crossings and permutations. In: Healy, P., Nikolov, N.S. (eds.) GD 2005. LNCS, vol. 3843, pp. 1–12. Springer, Heidelberg (2006)
3. Brandes, U., Dwyer, T., Schreiber, F.: Visualizing related metabolic pathways in two and a half dimensions (long paper). In: Liotta, G. (ed.) GD 2003. LNCS, vol. 2912, pp. 111–122. Springer, Heidelberg (2004)
4. Brandes, U., Wagner, D.: visone - analysis and visualization of social networks. In: Graph Drawing Software, pp. 321–340. Springer, Heidelberg (2003)

5. Demestrescu, C., Finocchi, I.: Breaking cycles for minimizing crossings. J. Exp. Algorithmics 6, 2 (2001)
6. Eades, P., Wormald, N.C.: Edge crossings in drawings of bipartite graphs. Algorithmica 11, 379–403 (1994)
7. Finocchi, I.: Layered drawings of graphs with crossing constraints. In: Wang, J. (ed.) COCOON 2001. LNCS, vol. 2108, pp. 357–367. Springer, Heidelberg (2001)
8. Forster, M.: Applying crossing reduction strategies to layered compound graphs. In: Goodrich, M.T., Kobourov, S.G. (eds.) GD 2002. LNCS, vol. 2528, pp. 276–284. Springer, Heidelberg (2002)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)
10. Garey, M.R., Johnson, D.S.: Crossing number is NP-complete. SIAM J. Algebraic Discrete Methods 4(3), 312–316 (1983)
11. Jünger, M., Mutzel, P.: 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. Journal of Graph Algorithms and Applications 1(1), 1–25 (1997)
12. Li, X.Y., Stallmann, M.F.: New bounds on the barycenter heuristic for bipartite graph drawing. Inf. Process. Lett. 82(6), 293–298 (2002)
13. Mehlhorn, K., Näher, S.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (1999)
14. Munoz, X., Unger, W., Vrto, I.: One sided crossing minimization is np-hard for sparse graphs. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 115–123. Springer, Heidelberg (2002)
15. Nagamochi, H.: An improved bound on the one-sided minimum crossing number in two-layered drawings. Discrete Comput. Geom. 33(4), 569–591 (2005)
16. Smith, B.S., Lim, S.K.: Qca channel routing with wire crossing minimization. In: GLSVSLI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI, pp. 217–220. ACM Press, New York (2005)
17. Stallmann, M., Brglez, F., Ghosh, D.: Heuristics, experimental subjects, and treatment evaluation in bigraph crossing minimization. J. Exp. Algorithmics 6, 8 (2001)
18. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Transactions on Syst. Man. and Cybernetics 11(2), 109–125 (1981)
19. Yamaguchi, A., Sugimoto, A.: An approximation algorithm for the two-layered graph drawing problem. In: Asano, T., Imai, H., Lee, D.T., Nakano, S.-i., Tokuyama, T. (eds.) COCOON 1999. LNCS, vol. 1627, pp. 81–91. Springer, Heidelberg (1999)

Fast Minimum-Weight Double-Tree Shortcutting for Metric TSP

Vladimir Deineko[1] and Alexander Tiskin[2]

[1] Warwick Business School, The University of Warwick, Coventry CV47AL, UK
[2] Dept. of Computer Science, The University of Warwick, Coventry CV47AL, UK

Abstract. The Metric Traveling Salesman Problem (TSP) is a classical NP-hard optimization problem. The double-tree shortcutting method for Metric TSP yields an exponential-sized space of TSP tours, each of which is a 2-approximation to the exact solution. We consider the problem of *minimum-weight double-tree shortcutting*, for which Burkard et al. gave an algorithm running in time $O(2^d n^3)$ and memory $O(2^d n^2)$, where d is the maximum node degree in the rooted minimum spanning tree (e.g. in the non-degenerate planar Euclidean case, $d \leq 4$). We give an improved algorithm running in time $O(4^d n^2)$ and memory $O(4^d n)$, which allows one to solve the problem on much larger instances. Our computational experiments suggest that the minimum-weight double-tree shortcutting method provides one of the best known tour-constructing heuristics.

1 Introduction

The Metric Travelling Salesman Problem (TSP) is a classical combinatorial optimization problem. We represent a set of n points in a metric space by a complete weighted graph on n nodes, where the weight of an edge is defined by the distance between the corresponding points. The objective of Metric TSP is to find in this graph a minimum-weight Hamiltonian cycle (equivalently, a minimum-weight tour visiting every node at least once). The most common example of Metric TSP is the planar Euclidean TSP, where the points lie in the two-dimensional Euclidean plane, and the distances are measured according to the Euclidean metric.

Metric TSP, even restricted to planar Euclidean TSP, is well-known to be NP-hard [10]. Metric TSP is also known to be NP-hard to approximate to within a ratio 1.00456, but polynomial-time approximable to within a ratio 1.5. Fixed-dimension Euclidean TSP is known to have a PTAS (i.e. a family of algorithms with approximation ratio arbitrarily close to 1) [3]; this generalises to any metric defined by a fixed-dimension Minkowski vector norm.

Two simple methods, double-tree shortcutting [14] and Christofides' [5,15], allow one to approximate the solution of Metric TSP within a factor of 2 and 1.5, respectively. Both methods build an Eulerian graph on the given point set, select an Euler tour of the graph, and then perform *shortcutting* on this tour by removing repeated nodes, until all node repetitions are removed. Thus, the minimum-weight tour is approximated by picking an element from a restricted

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 136–149, 2007.
© Springer-Verlag Berlin Heidelberg 2007

space of tours, namely the shortcuttings of a particular Euler tour of a particular Eulerian graph. Both the double-tree shortcutting and Christofides' methods belong to the class of *tour-constructing heuristics*, i.e. "heuristics that incrementally construct a tour and stop as soon as a valid tour is created" [8].

The two methods differ in the way the initial weighted Eulerian graph is constructed. In the tree shortcutting method, it is obtained by finding a minimum-weight spanning tree (MST), and then doubling every edge in the tree. In the Christofides method, the Eulerian graph is obtained by adding to the MST a minimum-weight matching built on the set of odd-degree nodes in the MST.

All tours obtained by the double-tree shortcutting (respectively, Christofides) method are guaranteed to have weight at most a factor of 2 (respectively, 1.5) higher than the minimum-weight tour. Indeed, this is already true for the weight of the original Euler tour, and in Metric TSP the shortcutting process cannot increase the tour weight.

While any shortcutting of the original Euler tour provides the approximation ratio 2 (respectively, 1.5), clearly, it is still desirable to find the minimum-weight tour among all possible shortcuttings. Given an Eulerian graph on a set of points, we will consider its *minimum-weight shortcutting*, i.e. the minimum-weight shortcutting across all possible Euler tours of this Eulerian graph. We shall correspondingly speak about *the minimum-weight double-tree shortcutting* and *the minimum-weight Christofides* methods.

The problem of finding the minimum-weight double-tree shortcutting is NP-hard for Metric TSP. Indeed, consider an instance of the Hamiltonian cycle problem, which can be regarded as an instance of Metric TSP with distances 1 and 2. Add an extra node connected to all other nodes by edges of weight 1, and take the newly added edges as the MST. It is easy to see that the minimum-weight shortcutting problem on the resulting instance is equivalent to the original Hamiltonian cycle problem. The minimum-weight double-tree shortcutting problem was also believed for a long time to be NP-hard for planar Euclidean TSP, until a polynomial-time algorithm was given by Burkard et al. [4]. In contrast, the problem of finding the minimum-weight Christofides shortcutting is NP-hard both for Metric TSP and planar Euclidean TSP [11].

In the rest of this paper, we will mainly deal with the *rooted MST*, which is obtained from the MST by selecting an arbitrary node as *the root*. In the rooted MST, the terms *parent*, *child*, *ancestor*, *descendant*, *sibling* all have their standard meaning. Let d denote the maximum number of children per node in the rooted MST. Note that in the Euclidean plane, the maximum degree of an unrooted MST is at most 6. Moreover, a node can have degree equal to 6, only if it is surrounded by six equidistant nodes; we can exclude this degenerate case from consideration by a slight perturbation of the input points. This leaves us with an unrooted MST of maximum degree 5. By choosing a node of degree less than 5 as the root, we obtain a rooted MST with $d \leq 4$.

The minimum-weight double-tree shortcutting algorithm by Burkard et al. [4] applies to the general Metric TSP, and runs in time $O(2^d n^3)$ and space $O(2^d n^2)$. In this paper, we give an improved algorithm, running in time $O(4^d n^2)$ and

memory $O(4^d n)$, We then describe our implementation of the new algorithm, which incorporates a couple of additional heuristics designed to speed up the algorithm and to improve its approximation quality. Computational experiments show that the approximation quality and running time of our implementation are among the best of all known tour-constructing heuristics.

2 The Algorithm

Preliminaries. Let G be a weighted graph representing the Metric TSP problem on n points. The double-tree method consists of the following stages:

- construct the minimum spanning tree of G;
- duplicate every edge of the tree, obtaining an n-node Eulerian graph;
- select an Euler tour of the double-tree graph;
- reduce the Euler tour to a Hamiltonian cycle by repeated *shortcutting*, i.e. replacing a node sequence a, b, c by a, c, as long as node b appears elsewhere in the current tour.

We say that a Hamiltonian cycle *conforms* to the doubled spanning tree, if it can be obtained from that tree by shortcutting one of its Euler tours. We also extend this definition to paths, saying that a path conforms to the tree, if it is a subpath of a conforming Hamiltonian cycle.

In our minimum-weight double-tree shortcutting algorithm, we refine the bottom-up dynamic programming approach of [4]. Initially, we select an arbitrary node r as the root of the tree. For a node u, we denote by $C(u)$ the set of all children of u, and by $T(u)$ the node set of the maximal subtree rooted at u, i.e. the set of all descendants of u (including u itself). For a set of siblings U, we denote by $T(U)$ the (disjoint) union of all subtrees $T(u)$, $u \in U$.

The characteristic property of a conforming Hamiltonian cycle is as follows: for every node u, upon entering the subtree $T(u)$, the cycle must visit all nodes of $T(u)$ before leaving the subtree, and must not re-enter the subtree afterwards. For an arbitrary node set S, we will say that a path through the graph *sweeps* S, if it visits all nodes of S consecutively in some order. In this terminology, a conforming Hamiltonian cycle must, for every node u, contain a subpath sweeping the subtree $T(u)$.

In the rest of this section, we denote the metric distance between u and v by $d(u, v)$. We use the symbol $\uplus$ to denote disjoint set union. For brevity, given a set A and an element a, we write $A \uplus a$ instead of $A \uplus \{a\}$, and $A \setminus a$ instead of $A \setminus \{a\}$.

Computing the solution weight (upsweep). The algorithm proceeds by computing minimum-weight sweeping paths in progressively increasing subtrees, beginning with the leaves and finishing with the whole tree $T(r)$. A similar approach is adopted in [4], where in each subtree, all-pairs minimum-weight sweeping paths are computed. In contrast, our algorithm only computes single-source minimum-weight sweeping paths originating at the subtree's root. This leads to substantial savings in time and memory.

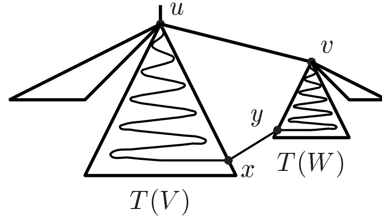


Fig. 1. Computation of $D^u_{V,W}(v)$

A non-root node $v \in C(u)$ is *active*, if its subtree $T(v)$ has already been processed, but its parent's subtree $T(u)$ has not yet been processed. In every stage of the algorithm, we choose *the current node* u, so that all children of u (if any) are active. We call $T(u)$ *the current subtree*. Let $V \subseteq C(u)$, $a \in T(V)$. By $D^u_V(a)$ we denote the weight of the shortest conforming path starting from u, sweeping subtree $u \uplus T(V)$, and finishing at a.

Consider the current subtree $T(u)$. Processing this subtree will yield the values $D^u_V(a)$ for all $V \subseteq C(u)$, $a \in T(V)$. In order to process the subtree, we need the corresponding values for all subtrees rooted at the children of u. More precisely, we need the values $D^v_W(a)$ for every child $v \in C(u)$, every subset $W \subseteq C(v)$, and every destination node $a \in T(W)$. We do not need any explicit information on subtrees rooted at grandchildren and lower descendants of u.

Given the current subtree $T(u)$, the values $D^u_V(a)$ are computed inductively for all sets V of children of u. The induction is on the size of the set V. The base of the induction is trivial: no values D^u_V exist when $V = \emptyset$.

In the inductive step, given a set $V \subseteq C(u)$, we compute the values $D^u_{V \uplus v}(a)$ for all $v \in C(u) \setminus V$, $a \in T(v)$, as follows. By the inductive hypothesis, we have the values $D^u_V(a)$ for all $a \in T(V)$. The main part of the inductive step consists in computing a set of auxiliary values $D^u_{V,W}(v)$, for all subsets $W \subseteq C(v)$. Every such value represents the weight of the shortest conforming path starting from node u, sweeping the subtree $u \uplus T(V)$, then sweeping the subtree $T(W) \uplus v$, and finishing at node v. Suppose the path exits the subtree $u \uplus T(V)$ at node x and enters the subtree $T(W) \uplus v$ at node y. We have

$$D^u_{V,W}(v) = \begin{cases} d(u,v) & \text{if } V = \emptyset,\ W = \emptyset \\ \min_{y \in T(W)} \left[d(u,y) + D^v_W(y) \right] & \text{if } V = \emptyset,\ W \neq \emptyset \\ \min_{x \in T(V)} \left[D^u_V(x) + d(x,v) \right] & \text{if } V \neq \emptyset,\ W = \emptyset \\ \min_{x \in T(V); y \in T(W)} \left[D^u_V(x) + d(x,y) + D^v_W(y) \right] & \text{if } V \neq \emptyset,\ W \neq \emptyset \end{cases} \tag{1}$$

(see Figure 1). The required values $D^v_W(y)$ have been obtained when processing subtrees $T(v)$ for $v \in C(u)$. Note that the computed auxiliary set contains $D^u_{V \uplus v}(v) = D^u_{V,C(v)}(v)$.

Now we can compute the values $D^u_{V \uplus v}(a)$ for all remaining $a \in T(v)$. A path corresponding to $D^u_{V \uplus v}(a)$ must sweep $u \uplus T(V)$, and then $T(v)$, finishing

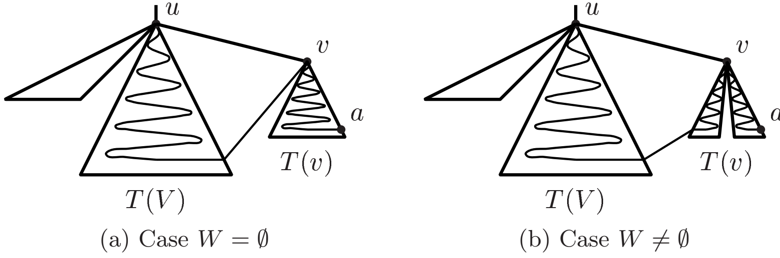


Fig. 2. Computation of $D^u_{V \uplus v}(a)$, $a \in T(v)$

at a. While in $T(v)$, the path will first sweep a (possibly single-node) subtree $v \uplus T(W)$, finishing at v. Then, starting at v, the path will sweep the subtree $v \uplus T(\overline{W})$, where $\overline{W} = C(V) \setminus W$, finishing at a. Considering every possible disjoint bipartitioning $W \uplus \overline{W} = C(V)$, such that $a \in T(\overline{W})$, we have

$$D^u_{V \uplus v}(a) = \min_{W \uplus \overline{W} = C(V) \colon a \in T(\overline{W})} \left[D^u_{V,W}(v) + D^v_{\overline{W}}(a) \right] \tag{2}$$

(see Figure 2).

We now have the values $D^u_{V \uplus v}(a)$ for all $a \in T(v)$. The computation (1)–(2) is repeated for every node $v \in C(u) \setminus V$. The inductive step is now completed.

The computation eventually reaches the root r of the tree, and establishes the values $D^r_S(a)$ for all $S \subseteq C(r)$, $a \in T(S)$. This includes the values $D^r_{C(r)}(a)$ for all $a \neq r$. The weight of the minimum-weight conforming Hamiltonian cycle can now be determined as

$$\min_{a \neq r} \left[D^r_{C(r)}(a) + d(a, r) \right] \tag{3}$$

Theorem 1. *The upsweep algorithm computes the weight of the minimum-weight tree shortcutting in time $O(4^d n^2)$ and space $O(2^d n)$.*

Proof. In computation (1), the total number of quadruples u, v, x, y is at most n^2 (since for every pair x, y, the node u is determined uniquely as the lowest common ancestor of x, y, and the node v is determined uniquely as a child of u and an ancestor of y). In computation (2), the total number of triples u, v, a is also at most n^2 (since for every pair u, a, the node v is determined uniquely as a child of u and an ancestor of y). For every such quadruple or triple, the computation is performed at most 4^d times, corresponding to 2^d possible choices of each of V, W. The cost of computation (3) is negligible. Therefore, the total time complexity of the algorithm is $O(4^d n^2)$.

Since our goal at this stage is just to compute the solution weight, at any given moment we only need to store the values $D^u_V(a)$, where u is either an active node, or the current node (i.e. the node for which these values are currently being computed). When u corresponds to an active node, the number of possible pairs u, a is at most n (since node u is determined uniquely as the root of an active subtree containing a). When u corresponds to the current node, the number of

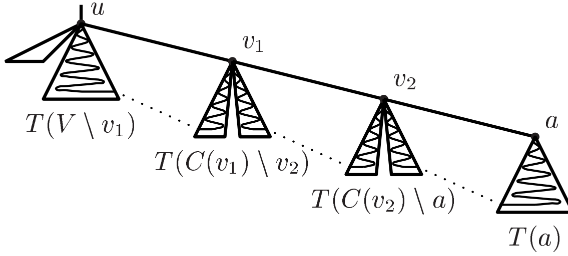


Fig. 3. Computation of $P^u_V(a)$, $a \in T(V)$, $k = 3$

possible pairs u, a is also at most n (since node u is fixed). For every such pair, we need to keep at most 2^d values, corresponding to 2^d possible choices of V. The remaining space costs are negligible. Therefore, the total space complexity of the algorithm is $O(2^d n)$. □

Computing the full solution (downsweep). In order to reconstruct the minimum-weight Hamiltonian cycle itself, we must keep all the auxiliary values $D^u_{V,W}(v)$ obtained in the course of the upsweep computation for every parent-child pair u, v. We solve recursively the following problem: given a node u, a set $V \subseteq C(u)$, and a node $a \in T(V)$, find the minimum-weight path $P^u_V(a)$ starting from u, sweeping subtree $u \uplus T(V)$, and finishing at a. To compute the global minimum-weight Hamiltonian cycle, it is sufficient to determine the path $P^r_{C(r)}(a)$, where r is the root of the tree, and a is the node for which the minimum in (3) is attained.

For any u, $V \subseteq C(u)$, $a \in T(V)$, consider the (not necessarily conforming or minimum-weight) path $u = v_0 \to v_1 \to v_2 \to \cdots \to v_k = a$, joining nodes u and a in the tree (see Figure 3). The conforming minimum-weight path $P^u_V(a)$ first sweeps the subtree $u \uplus T(C(u) \setminus v_1)$. After that, for every node v_i, $0 < i < k$, the path $P^u_V(a)$ sweeps the subtree $v_i \uplus T(C(v_i) \setminus v_{i+1})$ as follows: first, it sweeps a subtree $v_i \uplus T(W_i)$, finishing at v_i, and then, starting at v_i, it sweeps the subtree $v_i \uplus T(\overline{W}_i)$, for some disjoint bipartitioning $W_i \uplus \overline{W}_i = C(v_i) \setminus v_{i+1}$. Finally, the path $P^u_V(a)$ sweeps the subtree $T(a)$, finishing at a.

The optimal choice of bipartitionings can be found as follows. We construct a weighted directed graph with a source corresponding to node $u = v_0$, a sink corresponding to node $v_k = a$, and $k-1$ intermediate layers of nodes, each layer corresponding to a node v_i, $0 < i < k$. Each intermediate layer consists of at most 2^{d-1} nodes, representing all different disjoint bipartitionings of the set $C(v_i) \setminus v_{i+1}$. The source and the sink represent trivial bipartitionings $\emptyset \uplus (V \setminus v_1) = V \setminus v_1$ and $C(a) \uplus \emptyset = C(a)$, respectively. Every consecutive pair of layers (including the source and the sink) are fully connected by forward edges. In particular, the edge from a node representing the bipartitioning $X \uplus \overline{X}$ in layer i, to the node representing the bipartitioning $Y \uplus \overline{Y}$ in layer $i+1$, is given the weight $D^{v_i}_{\overline{X},Y}(v_{i+1})$. It is easy to see that an optimal choice of bipartitioning corresponds to the minimum-weight path from source to sink in the layered graph. This minimum-weight path can be found by a simple dynamic programming algorithm

(such as the Bellman–Ford algorithm, see e.g. [6]) in time proportional to the number of edges in the layered graph.

Let $W_1 \uplus \overline{W}_1, \ldots, W_{k-1} \uplus \overline{W}_{k-1}$ now denote the $k-1$ obtained optimal subtree bipartitionings. The k edges of the corresponding source-to-sink shortest path determine k edges (not necessarily consecutive) in the minimum-weight sweeping path $P_V^u(a)$. These edges are shown in Figure 3 by dotted lines. It now remains to apply the algorithm recursively in each of the subtrees $u \uplus T(V \setminus v_1)$, $v_1 \uplus T(W_1)$, $v_1 \uplus T(\overline{W}_1)$, $v_2 \uplus T(W_2)$, $v_2 \uplus T(\overline{W}_2)$, ..., $v_{k-1} \uplus T(W_{k-1})$, $v_{k-1} \uplus T(\overline{W}_{k-1})$, $T(a)$.

Theorem 2. *Given the output and the necessary intermediate values of the upsweep algorithm, the downsweep algorithm computes the edges of the minimum-weight tree shortcutting in time and space $O(4^d n)$.*

Proof. The construction of the auxiliary graph and the minimum-weight path computation runs in time $O(4^d k)$, where k is the number of edges in the tree path $u = v_0 \to v_1 \to v_2 \to \cdots \to v_k = a$ in the current level of recursion. Since the tree paths in different recursion levels are edge-disjoint, the total number of edges in these paths is at most n. Therefore, the time complexity of the downsweep stage is $O(4^d n)$.

By Theorem 1, the space complexity of the upsweep stage is $O(2^d n)$. In addition to the storage used internally by the upsweep stage, we also need to keep all the values $D_{V,W}^u(v)$. The number of possible pairs u, v is at most n (since node u is determined uniquely as the parent of v). For every such pair, we need to keep at most 4^d values, corresponding to 2^d possible choices of each of V, W. The remaining space costs are negligible. Therefore, the total space complexity of the algorithm is $O(4^d n)$. □

3 Heuristics and Computational Experiments

Despite the guaranteed approximation ratio of the double-tree shortcutting and Christofides methods, neither has performed well in previous computational experiments (see [7,13]). However, to our knowledge, none of the experiments exploited the minimum-weight shortcutting approach. In particular, Reinelt [13] compares 37 tour-constructing heuristics, including the depth-first double-tree algorithm and the Christofides algorithm, on a set of 24 geometric instances from the TSPLIB database [12]. Although most instances in this experiment are quite small (2000 or fewer points), they still allow us to make some qualitative judgement about the approximation quality of different heuristics. Double-tree shortcutting (without the minimum-weight improvement) turns out to have the lowest quality of all 37 heuristics, while the quality of the Christofides heuristic is somewhat higher, but still far from the top.

Intuitively, it is clear that the reason for the poor approximation quality of the two heuristics may be in the wrong decisions made in the shortcutting steps, especially considering that the overall number of alternative choices is typically exponential. This observation motivated us to implement the minimum-weight

double-tree shortcutting algorithm from [4]. It came as no surprise that this algorithm showed higher approximation quality than all the tour constructing heuristics in Reinelt's experiment. Unfortunately, Reinelt's experiment did not account for the running time of the algorithms under investigation. The theoretical time complexity of the algorithm from [4] is $O(2^d n^3)$; in practice, our implementation exhibited quadratic growth in running time on most instances. Both the theoretical and the practical running time were relatively high, which raised some justifiable doubts about the overall superiority of the algorithm.

As it was expected, the introduction of a new efficient minimum-weight tree shortcutting algorithm described in Section 2 significantly improved the running time in our computational experiments. However, this improvement alone was not sufficient for the algorithm to compete against the best existing tour-constructing heuristics. Therefore, we extended our algorithm by two additional heuristics, one aimed at increasing the algorithm's speed, the other at improving its approximation quality.

The first heuristic, aimed at speeding up the algorithm, is suggested by the well-known bounded neighbour lists [8, p. 408]. Given a tree, we define the *tree distance* between a pair of nodes a, b, as the number of edges on the unique path from a to b in the tree. Given a parameter k, the *depth-k list* of node u includes all nodes in the subtree $T(u)$ with the tree distance from u not exceeding k. The suggested heuristic is to limit the search across a subtree rooted at u in (1)–(2) to a depth-k list of u for a reasonably low value of k. Our experiments suggest that this approach improves the running time dramatically, while not having a significant effect on the approximation quality.

The second heuristic, aimed at improving the approximation quality, works by increasing the space of the tours searched by the double-tree method, in the hope of finding a better solution in the larger space. Let T be a (not necessarily minimum) spanning tree, and let $\Lambda(T)$ be the set of all tours conforming to the tree, i.e. the exponential set of all tours considered by the double-tree algorithm. Our goal is to construct a new tree T_1, such that its node degrees are still bounded by a constant, but $\Lambda(T) \subsetneq \Lambda(T_1)$. We refer to the new set of tours as an *enlarged tour neighborhood*.

Consider a node u in T, and suppose u has at least one child v which is not a leaf. We construct a new tree T_1 from T by applying the *degree increasing operation*, which makes node v a leaf, and redefines all children of v to be children of u. It is easy to check that any tour conforming to T also conforms to T_1. In particular, the nodes of $T(v)$, which are consecutive in any conforming tour of T, are still allowed to be consecutive in any conforming tour of T_1. Therefore, $\Lambda(T) \subseteq \Lambda(T_1)$. On the other hand, sequence w, u, v, where w is a child of v, is allowed by T_1 but not by T. Therefore, $\Lambda(T) \subsetneq \Lambda(T_1)$.

We apply the above degree-increasing heuristic as follows. Let D be a global parameter, not necessarily related to the maximum node degree in the original tree. The degree-increasing operation is performed only if the resulting new degree of vertex u would not exceed D. Note that when the maximum degree bound would be exceeded, this cannot be avoided by performing the degree

increasing operation partially: it would be wrong to reassign only some, instead of all, children of node v to a new parent. To illustrate this statement, suppose that v has two children w_1 and w_2, which are both leaves. Let w_2 be redefined as a new child of u. The sequence v, w_2, w_1 is allowed by T but not by T_1, since it violates the requirement for v and w_2 to be consecutive. Therefore, $\Lambda(T) \not\subseteq \Lambda(T_1)$.

Given a tree, the degree increasing operation is applied repeatedly to construct a new tree, obtaining an enlarged tour neighbourhood. In our experiments, we used breadth-first application of the degree increasing operation as follows:

Root the minimum spanning tree at a node of degree 1;
Let r' denote the unique child of the root;
Insert all children of r' into queue Q;
while queue Q is not empty **do**
  extract node v from Q;
  insert all children of v into Q;
  if $\deg(\mathit{parent}(v)) + \deg(v) \leq D$ **then**
    redefine all children of v to be children of $\mathit{parent}(v)$

Incorporating the above two heuristics, the minimum-weight double-tree algorithm from Section 2 was modified to take two parameters: the search depth k, and the degree-increasing limit D. We refer to the double-tree algorithm with fixed parameters k and D as a *double-tree heuristic* $DT_{D,k}$. We use DT without subscripts to denote the original minimum-weight double-tree algorithm, equivalent to $\mathrm{DT}_{1,\infty}$.

We compared experimentally the efficiency of the original algorithm DT with the efficiency of double-tree heuristics $\mathrm{DT}_{D,k}$ for two different search depths $k = 16, 32$, and for four different values for the maximum degree parameter $D = 1$ (no degree increasing operation applied), 3, 4, 5. The case $D = 2$ is essentially equivalent to $D = 1$, and therefore not considered. In our computational experiments we used the test data from DIMACS Implementation Challenge [8,2]. These are uniform random Euclidean instances with 1000 points (10 instances), 3162 points (five instances), 10000 points (three instances), 31623 and 100000 points (two instances of each size), 316228, 1000000, and 3168278 points (one instance of each size).

For each heuristic, we consider both its running time and approximation quality. We say that one heuristic *dominates* another, if it is superior in both these respects. The experimental results, presented in Table 1, clearly indicate that nearly all considered heuristics (excluding $\mathrm{DT}_{1,16}$) dominate plain DT. Moreover, all these heuristics (again excluding $\mathrm{DT}_{1,16}$) dominate DT on each individual instance used in the experiment. For further comparison of the double-tree heuristics with existing tour-constructing heuristics, we chose $\mathrm{DT}_{1,16}$ and $\mathrm{DT}_{5,16}$.

The main part of our computational experiments consisted in comparing the double-tree heuristics against the most powerful existing tour-constructing heuristics. As a base for comparison, we chose the heuristics analysed in [8], as well as two recent matching-based heuristics from [9]. The experiments were performed on a Sun Systems Enterprise Server E450.

Table 1. Results for DT and $DT_{D,k}$ on uniform Euclidean distances

Size	1000	3162	10K	31K	100K	316K	1M	3M
DT	7.36	7.82	8.01	8.19	8.39	8.40	8.41	–
$DT_{1,16}$	8.64	9.24	9.10	9.43	9.74	9.66	9.72	9.66
$DT_{3,16}$	6.64	6.97	7.04	7.37	7.51	7.53	7.55	7.50
$DT_{3,32}$	6.52	6.84	6.92	7.21	7.31	7.36	7.37	7.31
$DT_{4,16}$	6.00	6.27	6.39	6.69	6.82	6.87	6.85	–
$DT_{4,32}$	5.93	6.22	6.33	6.60	6.74	6.78	6.77	–
$DT_{5,16}$	5.67	5.91	5.97	6.27	6.43	6.51	6.47	–
$DT_{5,32}$	5.62	5.89	5.93	6.23	6.38	6.46	6.43	–

(a) Average excess over the Held–Karp bound (%)

Size	1000	3162	10K	31K	100K	316K	1M	3M
DT	0.18	1.56	15.85	294.38	3533	51147	156659	–
$DT_{1,16}$	0.04	0.14	0.47	1.57	5.60	20.82	101.09	388.52
$DT_{3,16}$	0.10	0.33	1.12	3.55	11.90	40.91	138.41	491.58
$DT_{3,32}$	0.18	0.69	2.45	7.56	25.46	82.99	269.73	935.55
$DT_{4,16}$	0.23	0.84	2.78	8.81	29.02	94.36	307.31	–
$DT_{4,32}$	0.45	2.00	6.93	22.11	74.70	236.33	744.50	–
$DT_{5,16}$	0.62	2.30	7.79	24.48	81.35	253.59	807.74	–
$DT_{5,32}$	1.11	5.74	20.73	65.96	224.34	695.03	2168.95	–

(b) Average normalised running time (s)

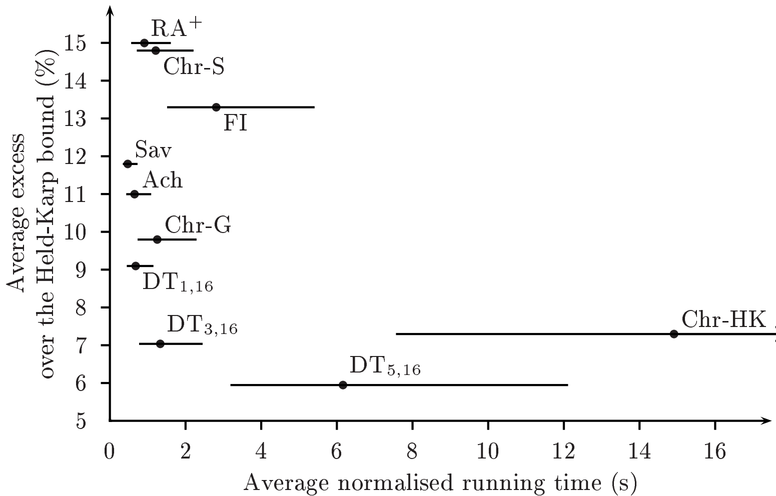


Fig. 4. Comparison between established heuristics and DT-heuristics on uniform Euclidean instances with 10000 points

Table 2 shows the results of these experiments. Abbreviations of the heuristics in the table follow [8,9]. As seen from the table, the average approximation

Table 2. Comparison between established heuristics and DT-heuristics on uniform Euclidean instances

Size	1000	3162	10K	31K	100K	316K	1M	3M
RA^+	13.96	15.25	15.04	15.49	15.43	15.42	15.48	15.47
Chr-S	14.48	14.61	14.81	14.67	14.70	14.49	14.59	14.51
FI	12.54	12.47	13.35	13.44	13.39	13.43	13.47	13.49
Sav	11.38	11.78	11.82	12.09	12.14	12.14	12.14	12.10
ACh	11.13	11.00	11.05	11.39	11.24	11.19	11.18	11.11
Chr-G	9.80	9.79	9.81	9.95	9.85	9.80	9.79	9.75
Chr-HK	7.55	7.33	7.30	6.74	6.86	6.90	6.79	–
MTS1	6.09	8.09	6.23	6.33	6.22	6.20	–	–
MTS3	5.26	5.80	5.55	5.69	5.60	5.60	–	–
$DT_{1,16}$	8.64	9.24	9.10	9.43	9.74	9.66	9.72	9.66
$DT_{5,16}$	5.67	5.91	5.97	6.27	6.43	6.51	6.47	–

(a) Average excess over the Held–Karp bound (%)

Size	1000	3162	10K	31K	100K	316K	1M	3M
RA^+	0.06	0.23	0.71	1.9	5.7	13	60	222
Chr-S	0.06	0.26	1.00	4.8	21.3	99	469	3636
FI	0.19	0.76	2.62	9.3	27.7	65	316	1301
Sav	0.02	0.08	0.26	0.8	3.1	21	100	386
ACh	0.03	0.12	0.44	1.3	3.8	28	134	477
Chr-G	0.06	0.27	1.04	5.1	21.3	121	423	3326
Chr-HK	1.00	3.96	14.73	51.4	247.2	971	3060	–
MTS1	0.37	2.56	17.21	213.4	1248	11834	–	–
MTS3	0.46	3.55	24.65	989.1	2063	21716	–	–
$DT_{1,16}$	0.04	0.14	0.47	1.57	5.60	20.82	101	389
$DT_{5,16}$	0.62	2.30	7.78	24.48	81.35	254	808	–

(b) Average normalised running time (s)

quality of $DT_{1,16}$ turns out to be higher than all classical heuristics considered in [8], except Chr-HK. Moreover, heuristic $DT_{1,16}$ dominates heuristics RA^+, Chr-S, FI, Chr-G. Heuristic $DT_{5,16}$ dominates Chr-HK. Heuristic $DT_{5,16}$ also compares very favourably with MTS heuristics, providing similar approximation quality at a small fraction of the running time. The above results show clearly that double-tree heuristics deserve a prominent place among the best tour-constructing heuristics for Euclidean TSP.

The impressive success of double-tree heuristics must, however, be approached with some caution. Although the normalised time is an excellent tool for comparing results reported in different computational experiments, it is only an approximate estimate of the exact running time. According to [8, page 377], "[this] estimate is still typically within a factor of two of the correct time". Therefore, as an alternative way of representing the results of computational experiments, we suggest a graph of the type shown in Figure 4, which compares the heuristics'

Table 3. Comparison between established heuristics and DT-heuristics on clustered Euclidean instances

Size	1000	3162	10K	31K	100K	316K
RA$^+$	12.84	13.88	16.08	15.59	16.22	16.33
Chr-S	12.03	12.79	13.08	13.47	13.50	13.45
FI	9.90	11.85	12.82	13.37	13.96	13.92
Sav	13.51	15.97	17.21	17.93	18.20	18.50
ACh	10.21	11.01	11.47	11.78	12.00	11.81
Chr-G	8.08	9.01	9.21	9.47	9.55	9.55
Chr-HK	7.27	7.78	8.37	8.42	8.46	8.56
MTS1	8.90	9.96	11.97	11.61	9.45	–
MTS3	8.52	9.5	10.11	9.72	9.46	–
$DT_{4,16}$	6.37	8.24	8.79	9.40	9.38	9.39
$DT_{5,16}$	5.72	7.17	7.92	8.32	8.46	8.42

(a) Average excess over the Held–Karp bound (%)

Size	1000	3162	10K	31K	100K	316K
RA$^+$	0.1	0.2	0.7	1.9	5.5	12.7
Chr-S	0.2	0.8	3.2	11.0	37.8	152.8
FI	0.2	0.8	2.9	9.9	30.2	70.6
Sav	0.0	0.1	0.3	0.9	3.4	22.8
ACh	0.0	0.2	0.8	2.1	6.4	54.2
Chr-G	0.2	0.8	3.2	11.0	37.8	152.2
Chr-HK	0.9	3.3	11.6	40.9	197.0	715.1
MTS1	0.78	4.19	45.09	276	1798	–
MTS3	0.84	4.76	49.04	337	2213	–
$DT_{4,16}$	0.2	0.87	3.16	9.55	34.43	120.3
$DT_{5,16}$	1.12	4.85	16.08	53.35	174	569

(b) Average normalised running time (s)

average approximation quality and running time on random uniform instances with 10000 points. A normalised time t is represented by the interval $[t/2, 2t]$. The relative position of heuristics in the comparison and the dominance relationships can be seen clearly from the graph. Results for other instance sizes and types are generally similar.

Additional experimental results for clustered Euclidean instances are shown in Table 3 (with $DT_{1,16}$ replaced by $DT_{4,16}$ to illustrate more clearly the overall advantage of DT-heuristics), and for TSPLIB instances in Table 4.

While we have done our best to compare the existing and the proposed heuristics fairly, we understand that our experiments are not, strictly speaking, a "blind test": we had the results of [8] in advance of implementing our method, and in particular of selecting the top DT-heuristics for comparison. However, we never consciously adapted our choices to the previous knowledge of [8], and we believe

Table 4. Comparison between established heuristics and DT-heuristics on geometric instances from TSPLIB: pr1002, pcb1173, rl1304, nrw1379 (size 1000), pr2392, pcb3038, fnl14461 (size 3162), pla7397, brd14051 (size 10K), pla33810 (size 31K), pla859000 (size 100K)

Size	1000	3162	10K	31K	100K
RA^+	17.46	16.28	17.78	19.88	17.39
Chr-S	13.36	14.17	13.41	16.50	15.46
FI	15.59	14.28	13.20	17.78	15.32
Sav	11.96	12.14	10.85	10.87	19.96
ACh	9.64	10.50	10.22	11.83	11.52
Chr-G	8.72	9.41	8.86	9.62	9.50
Chr-HK	7.38	7.12	7.50	6.90	7.42
MTS1	7.0	6.9	5.1	4.7	4.1
MTS3	6.2	5.1	4.0	2.9	2.7
$DT_{1,16}$	6.36	5.99	8.09	9.99	10.02
$DT_{5,16}$	6.13	5.58	7.65	8.98	9.30

(a) Average excess over the Held–Karp bound (%)

Size	1000	3162	10K	31K	100K
RA^+	0.1	0.2	0.8	2.2	5.6
Chr-S	0.1	0.2	1.8	3.9	31.8
FI	0.2	0.8	3.1	9.8	26.4
Sav	0.0	0.1	0.3	0.6	1.4
ACh	0.0	0.1	0.5	1.5	3.9
Chr-G	0.1	0.2	1.8	3.8	29.5
Chr-HK	0.7	2.2	9.7	50.1	177.9
MTS1	–	1.5	34.4	107.3	620.0
MTS3	–	2.1	42.4	135.4	1045.3
$DT_{1,16}$	0.3	0.9	4.1	18.4	49.3
$DT_{5,16}$	0.6	2.1	11.0	57.1	115.1

(b) Average normalised running time (s)

that any subconscious effect of this previous knowledge on our experimental setup is negligible.

4 Conclusions and Open Problems

In this paper, we have presented an improved algorithm for finding the minimum-weight double-tree shortcutting approximation for Metric TSP. We challenged ourselves to make the algorithm as efficient as possible. The improvement in time complexity from $O(2^d n^3)$ to $O(4^d n^2)$ (which implies $O(n^2)$ for the Euclidean TSP) placed the minimum-weight double-tree shortcutting method as a peer in the set of the most powerful tour-constructing heuristics. It is known that most powerful tour-constructing heuristics have theoretical time complexity $O(n^2)$, an in practice often exhibit near-linear running time. The minimum-weight double-tree method now also fits this pattern.

Our results should be regarded only as a first step in exploring new opportunities. Particularly, the minimum spanning tree is not the only possible choice of the initial tree. Instead, one can choose from a variety of trees, e.g. Held and Karp (1-)trees, approximations to Steiner trees, spanning trees of Delaunay graphs, etc. This variety of choices merits a further detailed exploration.

Our efforts invested into theoretical improvements of the algorithm, supported by a couple of additional heuristics, have borne the fruit: computational experiments with the minimum-weight double-tree algorithm show that it becomes one of the best known tour constructing heuristics. It appears that the double-tree method is well suited for local search heuristics based of transformations of trees and searching corresponding tour neighborhoods. One can easily imagine many

alternative tree transformation heuristics that could make our method even more powerful.

Acknowledgements

The authors thank an anonymous referee of a previous version of this paper, whose detailed comments helped to improve it significantly. The MST subroutine in our code is courtesy of the Concorde project [1].

References

1. Concorde TSP solver. http://www.tsp.gatech.edu/concorde
2. DIMACS TSP challenge. http://www.research.att.com/~dsj/chtsp
3. Arora, S.: Polynomial-time approximation schemes for Euclidean TSP and other geometric problems. Journal of the ACM 45, 753–782 (1998)
4. Burkard, R.E., Deineko, V.G., Woeginger, G.J.: The travelling salesman and the PQ-tree. Mathematics of Operations Research 23(3), 613–623 (1998)
5. Christofides, N.: Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon University (1976)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Electrical Engineering and Computer Science Series. The MIT Press and McGraw–Hill, second edition (2001)
7. Johnson, D.S., McGeoch, L.A.: The traveling salesman problem: a case study. In: Aarts, G., Lenstra, J.K. (eds.) Local Search in Combinatorial Optimisation, vol. 8, pp. 215–310. John Wiley & Sons, New York (1997)
8. Johnson, D.S., McGeoch, L.A.: Experimental analysis of heuristics for the STSP. In: Gutin, G., Punnen, A.P. (eds.) The Traveling Salesman Problem and Its Variations, ch 9, pp. 369–487. Kluwer Academic Publishers, Dordrecht (2002)
9. Kahng, A.B., Reda, S.: Match twice and stitch: a new TSP tour construction heuristic. Operations Research Letters 32, 499–509 (2004)
10. Papadimitriou, C.H.: The Euclidean traveling salesman problem is NP-complete. Theoretical Computer Science 4, 237–247 (1977)
11. Papadimitriou, C.H., Vazirani, U.V.: On two geometric problems related to the travelling salesman problem. Journal of Algorithms 5, 231–246 (1984)
12. Reinelt, G.: TSPLIB — a traveling salesman problem library. ORSA Journal on Computing 3(4), 376–384 (1991)
13. Reinelt, G.: The Travelling Salesman: Computational Solutions for TSP Applications. Springer, Heidelberg (1994)
14. Rosenkrantz, D.J., Stearns, R.E., Lewis II., P.M.: An analysis of several heuristics for the traveling salesman problem. SIAM Journal of Computing 6, 563–581 (1977)
15. Serdyukov, A.: On some extremal walks in graphs. Upravlyaemye systemy 17, 76–79 (1978)

A Robust Branch-Cut-and-Price Algorithm for the Heterogeneous Fleet Vehicle Routing Problem

Artur Pessoa[1], Marcus Poggi de Aragão[2], and Eduardo Uchoa[1]

[1] Departamento de Engenharia de Produção, Universidade Federal Fluminense, Brazil
{artur,uchoa}@producao.uff.br
[2] Departamento de Informática, Pontificia Universidade Católica do Rio de Janeiro, Brazil
poggi@inf.puc-rio.br

Abstract. This paper presents a robust branch-cut-and-price algorithm for the Heterogeneous Fleet Vehicle Routing Problem (HFVRP), vehicles may have various capacities and fixed costs. The columns in the formulation are associated to q-routes, a relaxation of capacitated elementary routes that makes the pricing problem solvable in pseudo-polynomial time. Powerful new families of cuts are also proposed, which are expressed over a very large set of variables. Those cuts do not increase the complexity of the pricing subproblem. Experiments are reported where instances up to 75 vertices were solved to optimality, a major improvement with respect to previous algorithms.

1 Introduction

This work considers a direct generalization of the classical Capacitated Vehicle Routing Problem (CVRP): the *Heterogeneous Fleet Vehicle Routing Problem* (HFVRP). Instead of assuming that all vehicles are identical, there is an availability of several vehicle types, with different characteristics. This generalization is very important to the operations research practice, most actual vehicle routing applications deal with heterogenous fleets. This problem has mostly been tackled by heuristics, some recent references include [5,9,12,13,15]. On the other hand, several authors explicitly commented about the difficulty of solving HFVRP instances to optimality or even of finding strong lower bounds [2,5,16,17]. We could not find any work claiming optimal solutions on any classical benchmark instance, even those with just 20 clients.

We define formally the problem. Let $G = (V, A)$ be a directed graph with vertices $V = \{0, 1, \ldots, n\}$ and $m = |A|$ arcs. Vertex 0 is the *depot*. The *clients* are the remaining vertices. Each client vertex i is associated with a positive integer demand $d(i)$. Depot demand $d(0)$ is defined as zero. There exists a fleet of K vehicle types, with integral capacities $\{C_1, \ldots, C_K\}$ and fixed costs $\{f_1, \ldots, f_K\}$. For each arc $a \in A$ there is a nonnegative travelling cost c_a. The HFVRP consists of finding a set of vehicle routes satisfying the following constraints: (i) each route, starting and ending at the depot, is assigned to a vehicle type k, (ii) each

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 150–160, 2007.
© Springer-Verlag Berlin Heidelberg 2007

client is included in a single route, and (iii) the total demand of all clients in a route is at most the capacity C_k of the chosen type. The goal is to minimize the sum of arc and fixed costs. We assume that $C_1 \leq \ldots \leq C_K$. For ease of notation, C will be used as a shorthand for C_K, the largest capacity.

The first lower bounds for the HFVRP were proposed by Golden et al. [6]. Stronger bounding schemes were proposed recently. Westerlund et al. [16] proposed an extended formulation and some valid cuts. This was only tested in a small sized instance with 12 clients. Choi and Tcha [2] produced lower bounds by using column generation. They are not good enough to build an effective branch-and-price algorithm. However, the approach of solving the restricted MIP provided by the columns generated at the root node proved to be a successful heuristic, obtaining the best known upper bounds on most benchmark instances from the literature. Yaman [17] performed a deep theoretical analysis of several different formulations and valid cuts. Practical experiments showed that this can lead to good lower bounds, but they are still not good enough for allowing the exact solution of the test instances.

This work departs from an extended HFVRP formulation to generate new valid cuts. It is shown that these cuts can be incorporated into a *Branch-Cut-and-Price* (BCP) algorithm in a *robust* way, i.e., without increasing the complexity of the pricing, as defined by Poggi de Aragão and Uchoa[11]. Computational experiments show that very strong lower bounds are obtained. In particular, instances with up to 75 clients can now be solved to optimality in reasonable times.

2 Formulations and Valid Cuts

There is an extended formulation for the HFVRP similar to the one given by Piccard and Queyranne [10] for the time-dependent TSP. Define $V_+ = \{1, \ldots, n\}$ as the set of all clients. Let binary variables x_a^d indicate that arc $a = (i, j)$ belongs to a route (of any vehicle type) and that the total demand of the remaining vertices in the route (including j) is exactly d. The arcs returning to the depot only have a variable with index $d = 0$. The *capacity-indexed* formulation follows:

$$\text{Minimize} \quad \sum_{a \in A} \sum_{d=0}^{C} \hat{c}_a^d x_a^d \tag{1a}$$

S.t.

$$\sum_{a \in \delta^-(i)} \sum_{d=1}^{C} x_a^d = 1 \qquad (\forall\, i \in V_+), \tag{1b}$$

$$\sum_{a \in \delta^-(i)} x_a^d - \sum_{a \in \delta^+(i)} x_a^{d-d(i)} = 0 \quad (\forall\, i \in V_+;\ d = d(i), \ldots, C), \tag{1c}$$

$$x_a^d \in \{0, 1\} \qquad (\forall\, a \in A;\ d = 1, \ldots, C), \tag{1d}$$

$$x_{(i,0)}^d = 0 \qquad (\forall\, i \in V_+;\ d = 1, \ldots, C) \tag{1e}$$

$$x_{(i,0)}^0 \in \{0, 1\} \qquad (\forall\, i \in V_+). \tag{1f}$$

Equations (1b) are in-degree constraints. Equations (1c) state that if an arc with index d enters vertex i then an arc with index $d - d(i)$ must leave i. Costs

$\hat{c}^d_a$, $a = (i, j)$, are equal to c_a if $i \neq 0$. Otherwise they are defined as $c_a + f_k$, where k is the index of smaller vehicle with capacity $C_k \geq d$. Variables with index distinct from 0 to the depot can be removed. Note that variables x^d_{ij} with $d > C - d(i)$ and with $d < d(j)$ can also be removed. To provide a more simple and precise notation when using the capacity-indexed variables, we define a directed multigraph $G_C = (V, A_C)$, where A_C contains arcs $(i, j)^d$, for each $(i, j) \in A$, $d = 1, \ldots, C - d(i)$, and $(i, 0)^0$, $i \in V_+$. In this context, it is assumed that $\delta^-(S)$ and $\delta^+(S)$ are the subsets of arcs in A_C, with any index, entering and leaving S. Denote by $\delta^-_d(S)$ and $\delta^+_d(S)$ the sets of arcs with index d entering and leaving S. For example, equations (1b) will be written as

$$\sum_{a^d \in \delta^-(i)} x^d_a = 1, \; (\forall\, i \in V_+).$$

Working directly with this formulation is only practical for small values of capacity, as there are $O(mC)$ variables and $O(nC)$ constraints. The capacity-indexed formulation can be naturally rewritten in terms of q-routes. A q-route [3] is a walk that starts at the depot vertex, traverses a sequence of client vertices with total demand at most equal to a given capacity, and returns to the depot. Some vertices may be visited more than once, therefore the set of q-routes strictly contains the set of actual routes. For each capacity C_k, $k = 1, \ldots, K$, suppose we have p_k possible q-routes. Define now q^{dj}_{ak} as the number of times arc a carrying exactly d units of capacity is traversed in the j-th q-route for vehicles with capacity C_k.

$$\text{Minimize} \quad \sum_{a^d \in A} \hat{c}^d_a x^d_a \tag{2a}$$

S.t.

$$\sum_{k=1}^{K} \sum_{j=1}^{p^k} q^{dj}_{ak} \lambda_{kj} - x^d_a = 0 \; (\forall\, a^d \in A_C), \tag{2b}$$

$$\sum_{a^d \in \delta^-(i)} x^d_a = 1 \qquad (\forall\, i \in V_+), \tag{2c}$$

$$\lambda_{kj} \geq 0 \qquad (k = 1, \ldots, K; \; j = 1, \ldots, p_k), \tag{2d}$$

$$x^d_a \in \{0, 1\} \qquad (\forall\, a^d \in A_C). \tag{2e}$$

It can be noted that equalities (1c) are implied by the definition of q-routes together with (2b). In fact (1) and (2) are equivalent in terms of their linear relaxation bounds. Eliminating the x variables and relaxing the integrality constraints, we get the Dantzig-Wolfe Master (DWM):

$$\text{Minimize} \quad \sum_{k=1}^{K} \sum_{j=1}^{p^k} \Big(\sum_{a^d \in A_C} q^{dj}_{ak} \hat{c}^d_a \Big) \lambda_{kj} \tag{3a}$$

S.t.

$$\sum_{k=1}^{K} \sum_{j=1}^{p^k} \Big(\sum_{a^d \in \delta^-(i)} q^{dj}_{ak} \Big) \lambda_{kj} = 1 \; (\forall\, i \in V_+), \tag{3b}$$

$$\lambda_{kj} \geq 0 \qquad (k = 1, \ldots, K; \; j = 1, \ldots, p). \tag{3c}$$

This LP has an exponential number of variables, but can be efficiently solved by column generation. The pricing subproblem amounts to finding minimum cost q-routes with given capacities C_k. This can be done in $O(n^2C)$ time by dynamic programming. However, in order to obtain stronger lower bounds one may also add cuts. A generic cut i over the extended variables

$$\sum_{a^d \in A_C} \alpha_{ai}^d x_a^d \geq b_i \tag{4}$$

can be also included in the DWM as

$$\sum_{k=1}^{K} \sum_{j=1}^{p^k} \Big(\sum_{a^d \in A_C} \alpha_{ai}^d q_{ak}^{dj} \Big) \lambda_{kj} \geq b_i . \tag{5}$$

Suppose that, at a given instant, we have a total of m constraints (including the in-degree constraints) in the DWM, the i-th constraint having with dual variable β_i. We define the reduced cost of arc a with index d as:

$$\bar{c}_a^d = c_a - \sum_{i=1}^{m} \alpha_{ai}^d \beta_i . \tag{6}$$

The resulting pricing subproblem, finding minimum cost q-routes with respect to capacity-indexed reduced costs $\bar{c}_a^d$ can still be solved in $O(n^2C)$ time, basically by the same dynamic programming algorithm. The above reformulation presents remarkable features. It allows the introduction of new cuts over the capacity-indexed variables, even for large values of C, without having to explicitly introduce any new variables and without changing the pricing subproblem.

2.1 Extended Capacity Cuts

For any set $S \subseteq V_+$, define $d(S) = \sum_{i \in S} d(i)$ and $\kappa(S) = \lceil d(S)/C \rceil$. The well-known Asymmetric CVRP cut $\sum_{a^d \in \delta^-(S)} x_a^d \geq \kappa(S)$ is also valid for the HFVRP. However, this *rounded capacity cut* is usually ineffective, since $\kappa(S)$, a lower bound on the number of routes that must enter S, is obtained by only considering the largest possible route capacity C. We introduce a family of stronger cuts over the capacity-indexed variables that are more suited for the HFVRP. For each vertex $i \in V_+$ the following balance equation is valid:

$$\sum_{a^d \in \delta^-(i)} d x_a^d - \sum_{a^d \in \delta^+(i)} d x_a^d = d(i) . \tag{7}$$

Summing the equalities (7) corresponding to each $i \in S$, we get the *capacity-balance equation over* S:

$$\sum_{a^d \in \delta^-(S)} d x_a^d - \sum_{a^d \in \delta^+(S)} d x_a^d = d(S) . \tag{8}$$

It can be noted that those equations are always satisfied by the solutions of (3) (translated to the x^d space by (2b)). Nevertheless, they can be viewed as the source of a rich family of cuts.

Definition 1. *An* Extended Capacity Cut *(ECC) over S is any inequality valid for $P(S)$, the polyhedron given by the convex hull of the 0-1 solutions of* (8).

The rounded capacity cuts could be derived only from the above definition: for a given S relax (8) to $\geq$, divide both sides by C and round coefficients up. Remember that $\delta^+(S)$ contains no arc with capacity C, so all such coefficients are rounded to zero. All coefficients corresponding to $\delta^-(S)$ are rounded to one. However, one may take advantage of the capacity-indices to get a *strengthened rounded capacity cut*:

$$\frac{\kappa(S)+1}{\kappa(S)} \sum_{a^d \in \delta^-(S)\,:\,d>d^*} x_a^d + \sum_{a^d \in \delta^-(S)\,:\,d\leq d^*} x_a^d - \frac{1}{\kappa(S)} \sum_{a^d \in \delta^+(S)\,:\,d\geq d'} x_a^d \geq \kappa(S)+1, \tag{9}$$

where $d^* = d(S) - C(\kappa(S)-1) - 1$ and d' is the smallest integer such that (i) $d' \geq d^* + 1 - \min_{i\in S}\{d(i)\}$ and (ii) $d' \geq C - d^*$. This is valid because if at least one route enters S with capacity $d \leq d^*$, we still need at least $\kappa(S)$ additional routes to cover S. Moreover, observe that each arc $a_1^{d_1} \in \delta^-(S)$ is associated to the next arc $a_2^{d_2} \in \delta^+(S)$ in the same route. Then, if at least one pair of associated arcs has $d_1 - d_2 \leq d^*$, we still need $\kappa(S)$ additional entering routes. If a route leaves S with $d \geq d'$, then (i) ensures that the corresponding arc entered S with demand greater than d^* and (ii) ensures that the capacity actually left in S by this route is not greater than d^*. So, the coefficient $-1/\kappa(S)$ reduces the total contribution of such pair of arcs to the left-hand side of (9) from $(\kappa(S)+1)/\kappa(S)$ to 1.

Many other kinds of ECCs can be derived. The *Homogeneous Extended Capacity Cuts* (HECCs) are a subset of the ECCs where all entering variables with the same capacity have the same coefficients, the same happening with the leaving variables. For a given set S, define aggregated variables y^d and z^d as follows:

$$y^d = \sum_{a^d \in \delta_d^-(S)} x_a^d \qquad (d = 1, \ldots, C), \tag{10}$$

$$z^d = \sum_{a^d \in \delta_d^+(S)} x_a^d \qquad (d = 0, \ldots, C). \tag{11}$$

The capacity-balance equation over those variables is:

$$\sum_{d=1}^{C} d y^d - \sum_{d=0}^{C} d z^d = d(S)\ . \tag{12}$$

For each possible pair of values of C and $D = d(S)$, we may define the polyhedron $P(C, D)$ induced by the integral solutions of (12). The inequalities that are valid for those polyhedra are HECCs. In Subsection 3.2 we illustrate how valid cuts can be derived and separated from that equality. Similar cuts have already been used with success on the Capacitated Minimum Spanning Tree Problem [14].

2.2 Triangle Clique Cuts

Let $S \subseteq V_+$ be a set of exactly three vertices. The triangle clique cuts are a family of cuts defined over the variables x_a^d, with $a^d = (i,j)^d \in A_C$ and $i,j \in S$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the compatibility graph where each vertex of $\mathcal{V}$ is a capacity-indexed arc $a^d = (i,j)^d \in A_C$ with $i,j \in S$. In this case, an edge $e = (a_1^{d_1}, a_2^{d_2})$ belongs to $\mathcal{E}$ if and only if $a_1^{d_1}$ and $a_2^{d_2}$ are *compatible*. There are four cases:

Case 1: if $e = ((i,j)^{d_1}, (i,k)^{d_2})$, then $e \notin \mathcal{E}$;
Case 2: if $e = ((i,j)^{d_1}, (k,j)^{d_2})$, then $e \notin \mathcal{E}$;
Case 3: if $e = ((i,j)^{d_1}, (j,k)^{d_2})$ and $d_1 \neq d_2 + d(j)$, then $e \notin \mathcal{E}$;
Case 4: if $e = ((i,j)^{d_1}, (j,k)^{d_2})$ and $d_1 = d_2 + d(j)$, then $e \in \mathcal{E}$;

For any independent set $I \subset \mathcal{V}$, it is clear that the following inequality is valid

$$\sum_{a^d \in I} x_a^d \leq 1. \tag{13}$$

This is a well-known clique cut. However $\mathcal{G}$ has a nice structure that can be explored to build a very efficient separation procedure, as will be shown in Subsection 3.2.

3 A Robust Branch-Cut-and-Price Algorithm

3.1 Column Generation

Recall that the reduced cost of a λ variable in DWM (3) is the sum of the reduced costs $\bar{c}_a^d$ of the arcs in the corresponding q-route. Those reduced costs are calculated using equations (6). The pricing subproblem of finding the q-routes yielding a variable with minimum reduced cost is NP-hard (it contains the capacitated shortest path problem), but can be solved in pseudo-polynomial $O(n^2 C_k)$ time. Since the reduced costs of the arcs that are not leaving the depot are independent of k, it is possible to price q-routes for every capacity C_k by making a single call (with $C_k = C$) to this dynamic programming algorithm. Houck et al. [7] and Christofides et al. [3] already noted that one can find q-routes without 2-cycles (subpaths $i \rightarrow j \rightarrow i$, $i \neq 0$) without changing this complexity. This immediately leads to a stronger formulation.

There is an important detail that must be taken into account in this column generation. Our HFVRP formulation is directed, but the benchmark instances from the literature are undirected. This should not be a problem, the directed problem is more general and contains the undirected case. However this leads to a symmetric cost structure, allowing two representations (the two possible arc orientations) for what is essentially the same q-route (or route). As a consequence, a convergence difficulty may appear. Cuts that are asymmetric regarding the arcs that enter or exit a subset of vertices may become not violated by simply changing the orientation of one or a few q-routes, without increasing the lower bound. This is the case for all the ECCs. We can deal with this difficulty

by requiring that, on undirected instances, the first client visited by a q-route must have a smaller identification than the last client. The modified algorithm has a worst case complexity of $O(n^3C)$, but several specific data structures can be used to improve the average case performance. In practice, this symmetry breaking strategy only introduces a small factor on the computing time.

3.2 Separation Routines

Let $\bar{\lambda}$ be a fractional solution of the DWM LP. This solution can be converted into a $\bar{x}$ solution over the capacity-indexed arc space using equations (2b). Violated cuts of form (4) can be separated and added to the DWM as (5).

Extended Capacity Cuts Our procedure starts by choosing candidate sets S. Those candidates include:

- All sets S up to cardinality 6 which are connected in the support graph of the fractional solution $\bar{x}$, i.e., the subgraph of G containing only the arcs a where some value $\bar{x}_a^d$ is positive. This connectivity restriction prevents an explosion on the number of enumerated sets. As proved in [14], if an ECC is violated over a set S composed of two or more disconnected components, there exists another violated ECC over one of those smaller components.
- The sets with cardinality larger than 6 that are inspected in the heuristic separation of rounded capacity cuts presented in [8]. The rationale is that if the rounded capacity cut is almost violated for a given set S, it is plausible that an extended capacity can be violated over that set. In particular, if the rounded capacity cut is violated, the ECC (9) will be certainly violated.

So, for each candidate set S, we first check if the strengthened rounded capacity cut (9) is violated. Then we try to separate HECCs from the equation (12) over S. In particular, we look for inequalities of the following form:

$$\sum_{d=1}^{C} \lceil rd \rceil y^d - \sum_{d=1}^{C-1} \lfloor rd \rfloor z^d \geq \lceil rd(S) \rceil \ , \tag{14}$$

where $0 < r \leq 1$. As discussed in [14], at most $0.3C^2$ rational multipliers r need to be tried in this integer rounding procedure.

Triangle Clique Cuts The separation procedure for the triangle clique cuts finds the independent set $I \subset \mathcal{V}$ in $\mathcal{G}$ that maximizes $\sum_{a^d \in I} \bar{x}_a^d$. Although the problem of finding a maximum-weight independent set is strongly NP-hard for general graphs, such an independent set can be found for $\mathcal{G}$ in a linear time by exploiting its very specific structure. It can be shown that $\mathcal{G}$ is actually a collection of chains, i.e., all its vertices have degree 1 or 2.

3.3 Branching with Route Enumeration

We branch over the edges of the undirected graph associated to G. We choose the pair $\{i, j\}$ such that the value $\bar{x}_{\{i,j\}} = \sum_{d=0}^{C}(\bar{x}_{(i,j)}^d + \bar{x}_{(j,i)}^d)$ is closer to 0.65.

On the left branch node we require that $\bar{x}_{\{i,j\}}$ must be 0, on the right branch node this must be greater or equal to 1. It can be shown that this is a valid branching strategy.

However, in order to improve the performance of our algorithm, we combine this traditional branching with a route enumeration technique inspired by the one described in Baldacci et al. [1]. When the integrality gap, the difference between the best known feasible solution and the current LP relaxation is sufficiently small, those authors found that it may be practical to enumerate all possible relevant elementary q-routes, i.e., all routes that have a chance of being part of the optimal solution. A route is non-relevant if (i) its reduced cost (with respect to the current values of (6)) is greater than the gap, or (ii) there exists another route visiting the same set of clients with smaller cost (with respect to the original arc costs c_a). If the number of relevant routes is not too large (say, in the range of tenths of thousands), the overall problem may be solved by feeding a general MIP solver with a set-partition formulation containing only those routes. If this set-partition can be solved, the optimal solution will be found and no branch will be necessary. Sometimes this leads to very significant speedups when compared to traditional branch strategies. However, it should be remarked that such route enumeration is an inherently exponential procedure. Its practical performance depends crucially on the gap value and it is also sensitive to the characteristics of the instance that is being solved. There is no guarantee that a combinatorial explosion will not happen, even on small sized instances.

Our hybrid strategy, devised to provide a robust approach, is to perform limited route enumerations (one for each vehicle type) after each branch-and-bound node is solved. This means that the enumeration is aborted if more than 40,000 relevant routes for some type k were already generated or if more than 600,000 states (partial non-dominated routes) are being kept by our dynamic programming algorithm. If those limits are not reached, a set-partition containing all relevant routes for each vehicle type is given to a MIP solver. If this can be solved in less than 50,000 branch-and-bound nodes, the original node is declared as *solved* and no branch will occur. Otherwise, the edge branching is performed and two more nodes must be solved. Of course, deeper nodes will have smaller gaps, at some point the enumeration will work. The overall effect may be a substantially smaller branch-and-bound tree. For example, where the traditional branching would need to reach depth 15, the hybrid strategy does not go beyond depth 7.

Our BCP also uses the route enumeration as an heuristic. If the actual gap g of a node is still too large and the limits are reached, we try the enumeration with a dummy gap of $g/2$. If this is still not enough, we try with $g/4$ and so on. If the enumeration now succeeds, we try an increased dummy gap of $(g/2+g/4)/2$. In short, we perform a sort of binary search to determine a dummy gap that will yield a set-partition of reasonable size. The solution of such MIPs may provide improved upper bounds.

Finally, we should remark that the route enumeration is a quite sophisticated dynamic programming procedure, several tricks are necessary to prevent an early explosion on the number of states.

4 Experiments

We tested the resulting algorithm for the HFVRP on the set of instances given in Golden et al. [6], which is the same used in the experiments reported by Yaman [17] and by Choi and Tcha [2]. We compare the lower bounds provided by the last two algorithms with our approach. The computational times from Yaman [17] correspond to experiments running on a Sun Ultra 12×400 MHz, while that from Choi and Tcha [2] are on a Pentium IV 2.6 GHz. Our code was executed on a Core 2 Duo running at 2.13 GHz with 2 GB of RAM. Linear programs and set-partition MIPs were solved by CPLEX 10.0. In all runs, we set the initial UB slightly higher than those reported in [2].

The comparison of root lower bounds are presented in Table 1, where the columns with header k and C contain the number of different types of vehicles and the largest capacity, respectively. It is worth noting that the bounds from Yaman [17], given in the columns **Yam.**, are obtained with a branch-and-cut on a flow formulation. The bounds from Choi and Tcha [2], columns **Choi**, come from a column generation algorithm on q-routes with 2 cycle elimination. Our algorithm for the HFVRP executes with 2 cycle elimination and route symmetry breaking. We report the previous best known upper bounds and the upper bound found by our algorithm. Values proved to be optimal are printed bold.

Additional BCP statistics are presented in Table 2, where the 2nd, 3rd and 4th columns contain the number of cuts of each type inserted in the root node. The headers **SRCC**, **Rd ECC** and **Clique** mean strengthened rounded capacity cuts, HECCs obtained by integer rounding and triangle clique cuts, respectively. The following 4 columns contain the execution times spent in the root node with column generation, LP solving, cut separation, and route enumeration +

Table 1. Comparison of Root Lower Bounds

Inst.	K	C	Yam. LB	Yam. Time(s)	Choi LB	Choi Time(s)	Our LB	Our Time(s)	Prev UB	Our UB
c20-3	5	120	912.40	–	951.61	0.4	961.03	3.2	961.03	**961.03**
c20-4	3	150	6369.51	–	6369.15	0.7	6437.33	6.6	6437.33	**6437.33**
c20-5	5	120	959.29	–	988.01	0.8	1000.79	14.3	1007.05	**1007.05**
c20-6	3	150	6468.44	–	6451.62	0.4	6516.47	8.4	6516.47	**6516.47**
c50-13	6	200	2365.78	397.1	2392.77	10.1	2401.01	152.8	2406.36	**2406.36**
c50-14	3	300	8943.94	175.6	8748.57	50.8	9111.69	335.9	9119.03	**9119.03**
c50-15	3	160	2503.61	142.8	2544.84	10.0	2573.38	352.2	2586.37	**2586.37**
c50-16	3	140	2650.76	142.1	2685.92	11.3	2706.21	358.3	2720.43	**2720.43**
c75-17	4	350	1689.93	1344.8	1709.85	206.9	1717.47	4729.2	1744.83	1734.52
c75-18	6	400	2276.31	1922.8	2342.84	70.1	2351.31	5154.3	2371.49	**2369.64**
c100-19	3	300	8574.33	1721.2	8431.87	1178.9	8648.93	2938.1	8661.81 *	8661.81
c100-20	3	200	3931.79	2904.0	3995.16	264.0	4005.41	3066.7	4039.49	-
avg gap			2.64%		1.61%		0.44%			

* The solution presented in [15] as having cost 8659.74 actually costs 8665.75.

Table 2. Branch-Cut-and-Price Statistics

Instance	# Root Cuts			Root Times				#	Total
	SRCC	Rd ECC	Clique	Col Gen	LP	Cut Sep	Enum +SP	Nodes	Time
c20-3	38	71	17	1.6	0.4	0.6	0.2	1	3.2
c20-4	119	140	14	2.5	1.1	2.5	0.3	1	6.6
c20-5	104	73	17	6.3	0.9	5.4	0.6	1	14.3
c20-6	129	72	3	3.7	0.6	3.3	0.3	1	8.4
c50-13	256	152	39	60.1	4.4	49.2	29.3	1	152.8
c50-14	1065	285	52	75.0	30.5	72.6	143.8	1	335.9
c50-15	346	252	43	49.5	8.6	42.6	239.9	1	352.2
c50-16	185	181	52	54.9	3.6	105.7	175.3	1	358.3
c75-17	431	151	77	313.4	24.8	193.7	3720.9	–	–
c75-18	231	214	56	264.6	5.5	265.9	4578.5	125	84286
c100-19	809	241	104	634.8	58.3	331.1	1893.2	–	–
c100-20	165	213	83	267.9	8.4	230.5	2496.4	–	–

set-partition solving, respectively. The last two columns show the total number of open nodes and the overall execution time by the BCP. No branching was required on instances with up to 50 clients, the number of relevant routes enumerated is small enough to solve the instance by solving a single set-partition problem. Instance c75-18 could be solved by the hybrid strategy, branching till the gap is small enough for the enumeration.

5 Comments

This text presented a RBCP for the HFVRP. The use of cuts defined over the extended formulation seems very promising and deserves further development. The ECCs here utilized, strengthened rounded capacity cuts and HECCs obtained by simple integer rounding can still be improved and better separated. Dash, Fukasawa, and Gunluk [4] have just characterized the facets of the polyhedron $P(C, D)$ induced by the integral solutions of (12). This may immediately lead to the separation of the strongest possible HECCs for a given set S. Moreover, the current choice of candidate sets for separation is still naive and could be improved. Another line of research is the development of cuts from the arc-indexed compatibility graph over sets with cardinality larger than 3. Odd-hole cuts and even more complex families of facet-defining cuts are already known to exist even for sets of cardinality 4 or 5.

All the new introduced families of cuts were found to be crucial in having a consistent algorithm. For example, if only the rounded HECC separation is disabled, instance c20-4 takes more than 1 hour to be solved by the same BCP code. Also, the enumeration technique has shown to be an effective heuristic when the current LB is sufficiently close to the optimal solution.

Acknowledgements. AP, MPA and EU were partially financed by CNPq grants 301175/2006-3, 311997/06-6 and 304533/02-5. AP and EU received support from Mestrado em Engenharia de Produção-UFF. The authors also thank the technical support by GAPSO Inc., Brazil.

References

1. Baldacci, R., Bodin, L., Mingozzi, A.: The multiple disposal facilities and multiple inventory locations rollon-rolloff vehicle routing problem. Computers and Operation Research 33, 2667–2702 (2006)
2. Choi, E., Tcha, D.-W.: A column generation approach to the heterogeneous fleet vehicle routing problem. Computers and Operations Research 34, 2080–2095 (2007)
3. Christofides, N., Mingozzi, A., Toth, P.: Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. Mathematical Programming 20, 255–282 (1981)
4. Dash, S., Fukasawa, R., Gunluk, O.: On the generalized master knapsack polyhedron. To appear LNCS - IPCO 2007 (2007)
5. Gendreau, M., Laporte, G., Musaraganyi, C., Taillard, E.D.: A tabu search heuristic for the heterogeneous fleet vehicle routing problem. Computers and Operations Research 26, 1153–1173 (1999)
6. Golden, B., Assad, A., Levy, L., Gheysens, F.: The fleet size and mix vehicle routing problem. Computers and Operations Research 11, 49–66 (1984)
7. Houck, D., Picard, J., Queyranne, M., Vegamundi, R.: The travelling salesman problem as a constrained shortest path problem. Opsearch 17, 93–109 (1980)
8. Lysgaard, J., Letchford, A., Eglese, R.: A new branch-and-cut algorithm for the capacitated vehicle routing problem. Mathematical Programming 100, 423–445 (2004)
9. Ochi, L., Vianna, D., Filho, A., Drummond, L.: A parallel evolutionary algorithm for the vehicle routing problem with heterogeneous fleet. Future Generation Computer Systems 14, 285–292 (1998)
10. Picard, J., Queyranne, M.: The time-dependant traveling salesman problem and its application to the tardiness problem in one-machine scheduling. Operations Research 26, 86–110 (1978)
11. de, Aragão, M.P., Uchoa, E.: Integer program reformulation for robust branch-and-cut-and-price. In: Wolsey, L. (ed.): Annals of Mathematical Programming in Rio, Búzios, Brazil pp. 56–61 (2003)
12. Renaud, J., Boctor, F.F.: A sweep-based algorithm for the fleet size and mix vehicle routing problem. European Journal of Operational Research 140, 618–628 (2002)
13. Taillard, E.D.: A heuristic column generation method for the heterogeneous fleet vrp. RAIRO 33, 1–14 (1999)
14. Uchoa, E., Fukasawa, R., Lysgaard, J., Pessoa, A., de, Aragão, M.P., Andrade, D.: Robust branch-cut-and-price for the capacitated minimum spanning tree problem over a large extended formulation. Mathematical Programming, on-line first (2007)
15. Wassan, N.A., Osman, I.H.: Tabu search variants for the mix fleet vehicle routing problem. Journal of the Operational Research Society 53, 768–782 (2002)
16. Westerlund, A., Göthe-Lundgren, M., Larsson, T.: Mathematical formulations of the heterogeneous fleet vehicle routing problem. In: ROUTE 2003, International Workshop on Vehicle Routing, pp. 1–2 (2003)
17. Yaman, H.: Formulations and valid inequalities for the heterogeneous vehicle routing problem. Mathematical Programming 106, 365–390 (2006)

Simple and Efficient Geographic Routing Around Obstacles for Wireless Sensor Networks

Olivier Powell* and Sotiris Nikoletseas**

Computer Engineering and Informatics Department of Patras University and
Research Academic Computer Technology Institute (CTI), Greece

Abstract. Geographic routing is becoming the protocol of choice for many sensor network applications. The current state of the art is unsatisfactory: some algorithms are very efficient, however they require a preliminary planarization of the communication graph. Planarization induces overhead and is thus not realistic for some scenarios such as the case of highly dynamic network topologies. On the other hand, georouting algorithms which do not rely on planarization have fairly low success rates *and* fail to route messages around all but the simplest obstacles. To overcome these limitations, we propose the `GRIC` geographic routing algorithm. It has absolutely no topology maintenance overhead, almost 100% delivery rates (when no obstacles are added), bypasses large convex obstacles, finds short paths to the destination, resists link failure and is fairly simple to implement. The case of hard concave obstacles is also studied; such obstacles are hard instances for which performance diminishes.

1 Introduction

Recent advances in micro-electromechanical systems (MEMS) have enabled the development of very small sensing devices called sensor nodes [1,2,3]. These are smart devices with sensing, data-processing and wireless transmission capabilities meant to collaboratively form *wireless sensor networks* (sensor nets) instrumenting the physical world by collecting, aggregating and propagating environmental information to regions of interest such as mobile users or fixed base stations possibly linked to a satellite or the Internet. Some applications imply deployment in remote or hostile environments (battle-field, tsunami, earth-quake, isolated wild-life island, space exploration program) to assist in tasks such as target tracking, enemy intrusion detection, forest fire detection or environmental or biological monitoring. Other applications imply deployment indoors or in urban or controlled environments, for example with the purpose of industrial supervising, indoor micro-climate monitoring (e.g. to reduce heating cost by detecting poor thermal insulation of buildings), smart-home applications, patient-doctor

* Work supported by the *Swiss National Science Foundation*, ref. PBGE2 - 112864.
** Work partially supported by the E.U. IST program num. IST-2005-15964 (AEOLUS).

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 161–174, 2007.
© Springer-Verlag Berlin Heidelberg 2007

health monitoring or blind and impaired assisting. Because of a few characteristics that differentiate them from otherwise similar ad hoc wireless nets such as MANETS, sensor nets raise a multitude of algorithmic challenges [4,5]. Among characteristic that make sensor nets very different [6] are strong resource limitations (energy, memory, processing power), high-density and size (which can be orders of magnitude greater than for other technologies) and the necessity to operate unattended under the constraint of environmental hazard.

Problem statement: We consider the problem of routing messages in a localized sensor net, a problem commonly called geographic routing (or georouting). We address the problem of finding a simple and efficient georouting algorithm which delivers messages with high success rate even in regions of low density (routing holes) and large communication blocking obstacles. The routing algorithms we allow ourselves to consider should be *lightweight*, *on demand* (thus making our algorithm all-to-all), *efficient* and *realistic*.

On the importance of geographic routing: According to [7], *"the most appropriate protocols* [for sensor nets] *are those that discover routes on demand using local, lightweight, scalable techniques, while avoiding the overhead of storing routing tables or other information that is expensive to update such as link costs or topology changes"*. This is due to the severe resource limitations of sensor devices and the high dynamics of the ad hoc networks they spontaneously establish. In view of this, geographic routing is very attractive [6,7]. The early and simple greedy georouting protocols [8] where messages are sent to the neighbour maximising progress towards the destination meet those idealistic requirements: the only information required to route is, assuming the nodes are localized, the destination of the message. One may wonder how realistic the assumption of localized nodes is, and to what extent it confines georouting to a specialised niche. Our point of view is that *"...geographic routing is becoming the protocol of choice for many emerging applications in sensor networks..."* [9] because location aware nodes are likely to be available since *"...in many circumstances, it is useful and even necessary for a node in a wireless sensor network to be aware of its location in the physical world. For example, tracking or event-detection functions are not particularly useful if the* [sensor net] *cannot provide any information* where *an event has happened"* [6]. Node localization is achievable through one of the many localization systems that use a combination of GPS like technology to localize a few beacon nodes followed by a distributed localization protocol [6,7,10,11,12]. Interestingly, it turns out that georouting can even be used when nodes are not location aware by using *virtual coordinates* as was proposed in [13].

State of the Art: The major problem of the early greedy georouting algorithms [8] is the so called *routing hole problem* [6,7,14] where messages get trapped in "local minimum" nodes which have no neighbours closer to the destination of the message than themselves. The incidence of routing holes increases as network density diminishes and the success rate of the greedy algorithm drops very quickly with network density. In order to bypass routing holes (and obstacles), very ingenious georouting algorithms have been developed. The most successful

ones are probably the celebrated GFG and GPSR algorithms [15,16] (as well as incremental improvements such as GOAFR [17], c.f. [18] for details) which have the very strong property of *guaranteeing successful routing* if the network is connected. GFG and GPSR are very similar and were, to our knowledge, developed independently. We use the encompassing term of *face routing* algorithms to refer to GFG, GPSR and their incremental successors. They all share, as a central idea, the use of a greedy propagation phase until the message reaches a local minimum. At this point a temporary *rescue mode* is used to escape the local minimum. The rescue mode uses (a variant of) the FACE algorithm originally described in [15] where messages are routed along the faces of the polygons of a planar subgraph of the communication graph. The use of a planar subgraph, which is necessary for face routing, is a crucial and restrictive characteristic: it implies that a graph planarization component has to be included in the routing algorithm. Until very recently, a major pitfall [19] of face routing algorithms was that no practical planarization algorithm was known: *"... all currently proposed geographic routing algorithms rely on idealized assumptions about radios and their resulting connectivity graphs* [...] *which are grossly violated by real radios* [...] *causing persistent failures in geographic routing, even on static topologies"* [20]. In a recent breakthrough paper [21] the first practical planarization algorithm with reasonable message overhead was proposed, *lazy cross-link removal* (LCR). Although reasonable, at least in static nets, LCR still induces a high topology maintenance overhead to discover impairing "cross-links", c.f. [21]. Another interesting approach is the BOUNDHOLE algorithm from [22] which uses the TENT rule to discover local minimum nodes and then "bounds" the contour of routing holes. Although it has a high overhead when compared to the algorithm we propose, the information gained during the contour discovery phase may be used for other application than routing such as path migration, information storage mechanisms and identification of regions of interest, c.f. [22] for details. Some other solutions are the probabilistic PFR, VTRP, LTP and CKN protocols [23,24,25,26]. These approaches are very different from face routing algorithms in the sense that, at the the cost of accepting lower success rates (particularly in low density networks), they induce very little topology maintenance overhead. Another drawback is that they fail to bypass large obstacles [27].

Our approach: To overcome the limitations of previous approaches we propose a new algorithm: **G**eo**R**out**I**ng around obsta**C**les (`GRIC`), pronounced "Greek" in reference to its design location: the University of Patras in Greece. The main idea of `GRIC` is to appropriately combine movement directly towards the destination (to optimize performance) with an inertia effect. Inertia forces messages to keep moving along the "current" direction and to closely follow the perimeter of obstacles in order to efficiently bypass them. Inertia permits to get out of many routing holes and to bypass some quite strongly blocking convex obstacles. To further improve our algorithm, a "right-hand rule" inspired component is used in combination with a virtual compass. The right-hand rule is a well known "wall follower" technique to get out of a maze [28] which is also used for face

routing. However, unlike face routing algorithms, GRIC has the advantage of using the right-hand rule on the complete communication graph, thus eliminating the planarization phase overhead. The right-hand rule permits to route messages around large obstacles, not only convex but also concave. It is useful even in the absence of obstacles, making the success rate of GRIC close to 100% even for very low density networks. We implement our algorithm and comparatively evaluate its performance against those of other representative algorithms (greedy, LTP and FACE). We focus on two performance measures: success rate and hop count. We study the impact on performance of several types of obstacles (both convex and concave) and representative regimes of network density.

Strengths of our approach: GRIC is very simple (for example when compared to face routing relying on LCR planarization) and thus easy to implement. It has a very high success rate, even in the case of low density networks. It is capable of bypassing large emission blocking obstacles (although for the hardest obstacles performance decreases with network density) using a short path, close to optimal in the absence of global knowledge of the network. It is particularly suitable for highly dynamic networks where links go up and down, e.g. because of environmental fluctuation or network congestion. This follows from the fact that, for a start, GRIC has absolutely no topology maintenance overhead (the only information required, at the node level, is a list of outbound neighbours) and from the fact that, as shown in our experiments, GRIC is not only robust when confronted with link failure: it also has the surprising property of actually *performing better when confronted to limited link instability*. To our knowledge, the near 100% success rate of GRIC (without obstacles) and its effective obstacle avoidance property is unique among lightweight routing protocols. It also offers a competitive alternative to face routing, probably with a different dedicated application niche: GRIC would be preferred for highly dynamic networks whereas face routing may be preferred in the case of more stable networks where the planarization overhead is paid off over time if the topology is static, implying that planarization does not need to be recomputed frequently. As a consequence, we feel that GRIC considerably improves the state of the art of geographic routing.

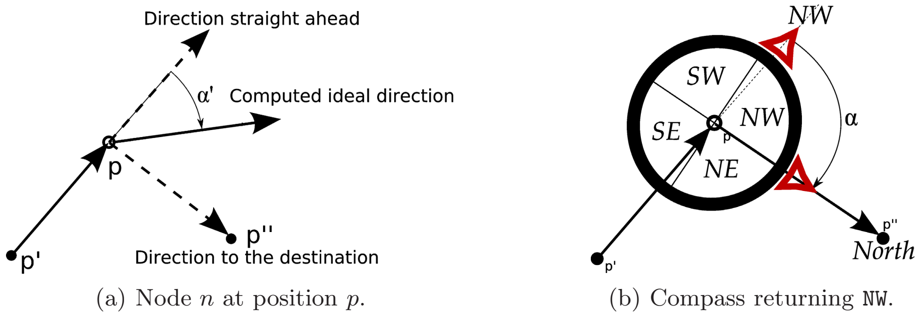
2 The GRIC Algorithm

Sensor Net Model: When a node needs to route a message according to the GRIC algorithm, it needs some network topology information. More precisely, nodes should be aware of their 1-hop away outbound neighbours, as well as their coordinates. In mathematical language, this is equivalent to assuming a *directed dynamic communication graph* (i.e. connectivity can change over time) embedded in the Euclidean plane. Although this may seem quite abstract at first sight, it is in fact very realistic: GRIC is a network layer protocol, it therefore relies on the data-link, MAC and physical layers. Many different MAC and data-link protocols exist, and although the study of the impact of different possible combinations is beyond the scope of this paper, most of them would provide, at the network layer, the level of abstraction we assume in this paper, c.f. [18] for

details. A final minor assumption is that we allow messages to piggy-back $\mathcal{O}(1)$ bits of information encoding the position of the last node visited, the position of the targeted message destination and a mark-up flag.

Overview: Like face routing algorithms , GRIC uses two different routing modes: a normal mode called *inertia mode* and a *rescue mode*. Intuitively, the inertia mode is used when the message makes progress towards the destination, and the rescue mode when it is going away from the destination. The inertia mode is inspired by physics and we use it to control the trajectory of messages inside the network. Messages are "attracted" to their destination but also have an incentive to follow the "straight line", like a celestial body is attracted in a planet system. The rescue mode adds a right-hand rule component to the inertia mode. The right hand-rule is a "wall follower" technique to get out of a maze [28], also used by face routing algorithms. GRIC combines it with inertia to bypass complex obstacles by following their contour.

Routing with inertia: We consider a node n at position p receiving a message m. n needs to take a routing decision for m. First, n reads the information piggy-backed on m to learn p' and p'', the position of the node which sent m to n and the destination position of m respectively, as illustrated in figure 1(a). Next, n computes $v_{\text{prev}} = p - p'$ and $v_{\text{dest}} = p'' - p$. If attached at position

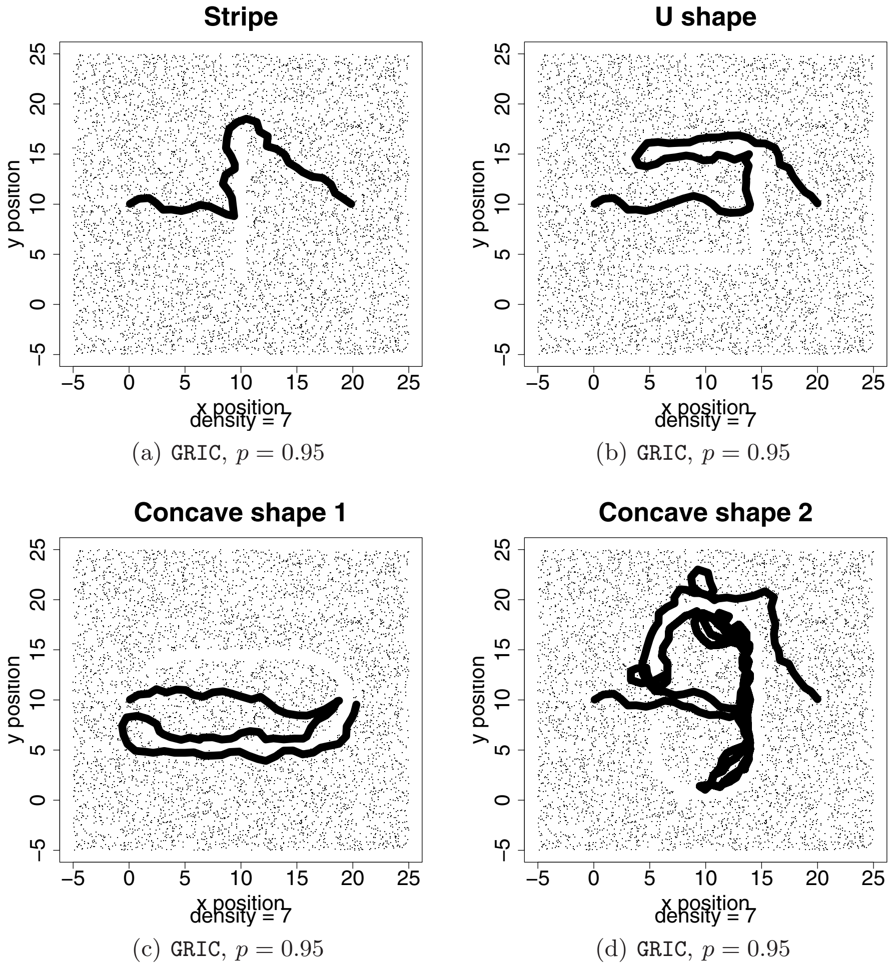


(a) Node n at position p.

(b) Compass returning NW.

Fig. 1. The compass device

p, v_{prev} is a vector pointing in the previous direction travelled by m whereas v_{dest} is a vector pointing in the direction of m's destination. Using elementary trigonometry, n computes the angle between v_{prev} and v_{dest}, which is uniquely defined if we allow only values in $[-\pi, \pi[$. n will try to send m in a computed ideal direction $v_{\text{ideal}} = \mathcal{R}_{\alpha'} \cdot v_{\text{prev}}$, where $\mathcal{R}_{\alpha'}$ is a rotation matrix of angle α' defined by $\alpha' = -\beta\pi$ if $\alpha < -\beta\pi$, $\alpha' = \beta\pi$ if $\alpha > \beta\pi$, $\alpha' = \alpha$ otherwise. The parameter β ranges in $[0, 1]$. If $\beta = 1$, $v_{\text{ideal}} = v_{\text{dest}}$ and inertia routing is equivalent to the greedy routing algorithm. At the other extreme when $\beta = 0$, $v_{\text{ideal}} = v_{\text{pev}}$ and the ideal direction is equal to the previous direction: inertia is maximal. The inertia factor can thus be controlled by adjusting β. In our simulations, setting $\beta = \frac{1}{6}$ proved to be a good choice for practical purposes. The final routing decision of n



(a) GRIC, $p = 0.95$ (b) GRIC, $p = 0.95$

(c) GRIC, $p = 0.95$ (d) GRIC, $p = 0.95$

Fig. 2. Typical behavior for different obstacle shapes

is to send m to its neighbour n_2 maximizing progress towards the ideal direction v_{ideal}, i.e. n_2 maximizes the scalar product $\langle v_{\text{ideal}} | pos(n_2) - p \rangle$.

Routing around obstacles: Our experiments show that inertia routing bypasses routing holes with high probability and routes messages around some large convex obstacles such as the one in figure 2(a). It is therefore used as the normal routing mode of GRIC. However, more complex concave obstacles as in figures 2(b),2(c) and 2(d) cannot be bypassed by inertia routing and we therefore add a *rescue mode* to GRIC. The first difficulty is to know *when* to switch to rescue mode. A *virtual compass device* and the use of a flag fulfill this purpose. We keep notations of the previous section and consider that node n receives a message m for which it needs to take a routing decision and describe below the steps required to implement the obstacle avoidance feature of GRIC.

• *The compass device:* n considers the north to be p'', i.e. the destination of m, and it wants to know what cardinal direction describes the last hop of m: north-west, north-east, south-west or south-east, c.f. figure 1(b). The answer depends on α: the virtual compass indicates `SW` if $\alpha \in [-\pi, -\pi/2[$, `NW` if $\alpha \in [-\pi/2, 0[$, `NE` if $\alpha \in [0, \pi/2[$ and `SE` otherwise.

• *The flag:* Intuitively, when m is being routed around an obstacle the flag should be up, otherwise it should be down. We metaphorically consider a walker following the path along which m is routed. When the walker follows the path and assuming m stays close to the obstacle's contour, two cases can occur: the obstacle's perimeter is either on the right or on the left of the walker. When this is so, we say the message is routed around the obstacle according to the right or left-hand rule respectively and `GRIC` acknowledges it by raising the flag and tagging it with `SW` or `SE` respectively. Formally, when n receives m, it starts by adjusting the flag's value in the following way. If the flag is down, the algorithm looks at the compass. If the compass points north, the flag stays down. However, if the compass points south, *the flag is raised.* and tagged with `SW` or `SE` respectively. If the flag is up, n has two options: leave the flag up (without changing the tag), or put the flag down. The flag goes down only if it was `SW`-tagged and the compass points `NW`, or if the flag was `SE`-tagged while the compass points `NE`.

• *Mode selection:* n receives the message m and it has to take a routing decision. The decision is taken in steps. First, n adjusts the flag according to the procedure described above. Next, it chooses whether to operate in the normal mode or whether it should switch to rescue mode. Only then will the routing decision be made and we shall soon describe how, but we first describe how to choose the routing mode. When the flag is down the normal mode is always used. When the flag is up, it depends. Suppose the flag is up and for the sake of simplicity let us assume it is `SW`-tagged, the other case being symmetric. Note that by case assumption this implies that the compass does not point `NW`, since otherwise the flag would be down. Also, in this case, intuitively `GRIC` tries to route around the obstacle using the right-hand rule. We next have to consider two cases. If the compass points `SW`, recalling the definition of v_{prev}, v_{ideal} and α, it is easy to see that by case assumption v_{ideal} is obtained by applying to v_{prev} a rotation of angle α' with α' in $[-\pi/6, 0]$, c.f. figure 1. In other words, inertia routing gives the message an incentive to turn to the right. This is consistent with the right-hand rule and `GRIC` thus chooses the normal routing mode. If the message points `SE` or `NE`, a similar reasoning shows that α' is in $[0, \pi/6]$ and that inertia routing will give an incentive for the message to turn left. However, this will get the message away from the obstacle (in the expected case where the obstacle's contour is indeed closely followed and kept on the right of the message). This is contrary to the right-hand rule idea and therefore `GRIC` will *switch to rescue-mode.*

• *The rescue mode:* By case assumption, the flag is up and we assume without loss of generality the tag to be `SW`, i.e. the right-hand rule applies. The

selection procedure previously described chooses rescue mode when inertia routing would give the message an unwanted incentive to turn left by computing an α' value in $[0, \pi/6]$. Intuitively, the rescue mode simply inverts the rotation angle α' in the following way: let $\alpha_2 = -sign(\alpha)(2\pi - |\alpha|)$. v_{ideal} is then defined by $v_{\text{ideal}} = R_{\alpha'_2} \cdot v_{\text{prev}}$, where $\alpha'_2 = \beta\alpha_2$ and where β is the same inertia conservation parameter as the one used for inertia routing. Putting all things so far together, `GRIC` is formally described by the (non-randomized version of) algorithm 1.

Algorithm 1. `GRIC`, running on the node n which is at position $pos(n)$.

1: **if** the flag is down *and* compass indicates `SW` (or `SE`) **then**
2: raise the flag and tag it with `SW` (or `SE` respectively)
3: **else if** the flag is up and the tag is `SW` (or `SE`) **then**
4: lower flag if the compass points `NW` (or `NE` respectively)
5: Decide if mode should be *normal* or *rescue* {c.f. "Mode selection" subsection}
6: **if** $mode = normal$ **then**
7: $\gamma := \alpha'$ {c.f. "Routing with inertia" section}
8: **else if** $mode = rescue$ **then**
9: $\gamma := \alpha'_2$ {c.f. "Rescue mode" subsection}
10: $v_{\text{ideal}} := R_\gamma \cdot v_{\text{prev}}$
11: Let $\mathcal{V}$ be the set of neighbours of n and let $\mathcal{V}$ be an empty set.
12: **if** running the random or non-random version of `GRIC` **then**
13: set $p = 0.95$ or $p = 1$ respectively
14: **for all** $v \in \mathcal{V}$ **do**
15: add v to $\mathcal{V}'$ with probability p
16: Send the message to the node $n_2 \in \mathcal{V}'$ maximizing $\langle v_{\text{ideal}} | pos(n_2) - pos(n) \rangle$

Randomization and robustness: While designing `GRIC`, we decided to test its robustness when confronted tp link instability. We found that *it performs better in the case of limited link instability.* Although surprising and perhaps counterintuitive, this feature of `GRIC` is easy to understand. Indeed, routing failure is likely to occur when a message deterministically starts looping around a local minimum. A small random perturbation breaks `GRIC`'s determinism and messages eventually escape the loop. This is a very nice property of `GRIC` which also suggests an easy way to improve its behavior in the context of stable networks: each time n needs to take a routing decision for a message m it starts by temporarily and randomly discarding each of its outbound links with probability p, c.f. line 12 of algorithm 1. For practical purposes experiments show the choice of $p = 0.95$ to be good. Further decreasing the value of p, to the best of our understanding, is tantamount to decreasing node density and thus performance slowly decreases.

3 Experiments

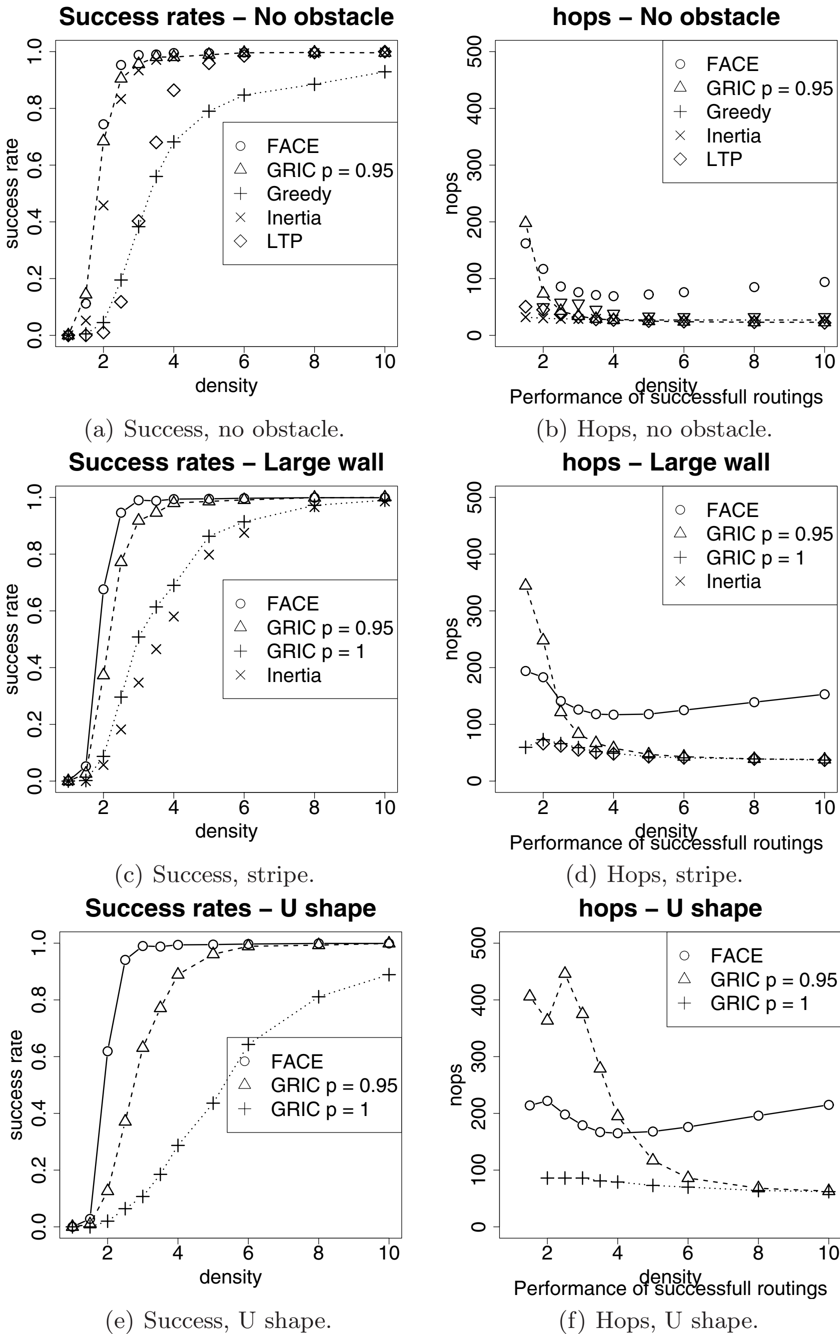
We validate the performance of `GRIC` through extensive experiments. A single experiment consists of randomly deploying a sensor net. A message is then

generated by a single node. The message has a destination, and the network tries to route the message to it. The experiment is successful if the message reaches its destination before a given timeout value, it is deemed to have failed otherwise. To verify the quality of successful outcomes, we measure the length in hops of the path that leads the message from source to destination. As a first experimental validation of `GRIC` and for practical purposes we resolve ourselves to simulation rather than experimenting with real sensor nets: even small size sensor nets are still quite prohibitively expensive and choosing and implementing a full protocol stack (MAC and data-link) on top of which our network layer algorithm operates implies a substantial amount of work which we delay for possible future work.

Simulation platform: We developed a high-level simulation platform using the Ruby programming language. As was previously explained, the `GRIC` algorithm is a network layer algorithm assuming a list of reliable collision free data-links to be made available by lower protocol stack layers. This assumption is weak since most physical/MAC/data-link suites would indeed provide this level of abstraction. We choose as a communication model the unit disc graph. Arguably, this model is the most commonly used for sensor net simulations. However, we acknowledge that this choice is not completely satisfactory and that more realistic communication graph models would be more appropriate. Defining a reasonable model suitable for simulation purposes is a challenging task. To our knowledge, only recently did the research community start to investigate this problem [29,30] and defining such a model is beyond the scope of this paper. Nevertheless, because our algorithm is robust in the presence of link failure, because it requires only one-hop-away neighbourhood discovery, because it does not even require links to be symmetric and because it implies *absolutely no topology maintenance*, we are confident that the unit disc communication graph is good enough to give a reasonable approximation of the behavior `GRIC` will have in real sensor nets.

Simulation details: We deploy randomly and uniformly N sensor nodes in the region of the Euclidean plane defined by $\{(x,y) \in \mathbb{R}^2 | -5 \leq x, y \leq 25\}$. The density of the network d is defined as $N/900$. Using the unit disc graph model, the expected number of neighbours per node is thus close to $d \cdot \pi$. A message m is generated at the point $(0, 10)$ and attached to the closest node. The destination of m is $(20, 0)$. m is propagated in the network according to the routing protocol considered. The experiment is considered successful if m gets within distance 1 of its destination (there may not be a node at the exact destination of the message). The outcome is a failure if m does not reach its destination in less than N steps. For precaution, we also consider the outcome to be failure if m approaches within distance 1 from the border of the network.

Preliminary results: We first consider the case where no obstacle is added to the network. We considered the FACE algorithm of [15]. FACE is not the most competitive algorithm in the face routing family in terms of path length, because it always runs in rescue mode. However, like all face routing algorithms it has the very strong "guaranteed delivery" property to always route a message

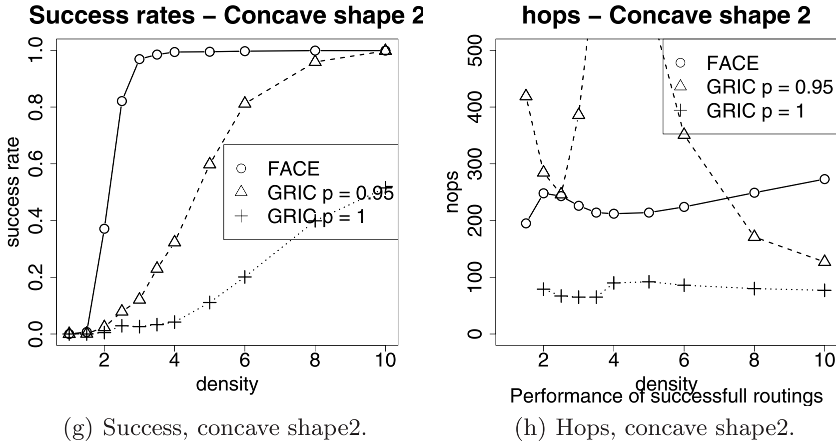


(a) Success, no obstacle. (b) Hops, no obstacle.

(c) Success, stripe. (d) Hops, stripe.

(e) Success, U shape. (f) Hops, U shape.

Fig. 3. Summary of simulation results



(g) Success, concave shape2.

(h) Hops, concave shape2.

Fig. 3. (*Continued*)

to its destination if a path exists from the source. In view of this, figure 3(a) reveals the following: in the absence of obstacles, the random version of `GRIC` (with $p = 0.95$) has almost 100% success rate even for low densities where the network is disconnected with high probability since the success rate is very close to that of FACE. We also observe in figure 3(b) that the path length of `GRIC` is close to optimal since it competes with greedy routing which is known to find very short paths, c.f. [18]. The performance of inertia routing is less than for `GRIC` (and slightly less than `GRIC` with $p = 1$, but for clarity we do not show this on the graph, c.f. [18] for details), but is still quite good and it outperforms the LTP protocol of [25]. (LTP uses a limited backtracking, we allowed LTP a maximum of 5 consecutive backtracking steps).

Obstacles: We consider four different types of large communication blocking obstacles.

•*Large wall:* The first obstacle is convex, c.f. figure 2(a). The random version of `GRIC` bypasses this large convex obstacle with high probability, c.f. figure 3(c). In terms of network design, it makes no sense to deploy a sensor net that has high probability of being disconnected and thus of being non-operational. We see from the performance of FACE that this critical density is around $d = 3$, and even for such a low density the performance of `GRIC` is very good, both in terms of success rate and path length. It is interesting to see that in the presence of this large obstacle, the randomization ($p = 0.95$) implies a high improvement over the deterministic version ($p = 1$), which not only implies that `GRIC` is resistant to link instability, it actually performs better in the presence of limited link instability. Interestingly, although not competitive we see that inertia is capable of bypassing the obstacle when the density is not too low. In [27], LTP was shown to have a good obstacle avoidance property. However, LTP never routes around a large wall such as the one we consider in this work (thus LTP is not included on plots). Therefore, to the best of our knowledge

GRIC outperforms other lightweight georouting protocols, both in the presence and absence of obstacles. In figure 3(d) we verify that path length is kept very low, considering that the message has no a priori knowledge of the network and discovers the obstacle only when reaching it.

• *U shape obstacle:* First of all, using a rule of thumb when looking in figures 2(b) and in figure 3(f) shows that the deterministic version of GRIC (i.e. when $p = 1$) rarely routes messages successfully to their destination but when it does so it uses a near optimal path. In light of this observation, we conclude that GRIC ($p = 0.95$) *bypasses this hard concave obstacle* and *uses a short path*. However, the performance is only good when the network density is around 5 and higher. This is a medium density for sensor nets since, again, a density below 3 is probably not acceptable in terms of network design since it yields a disconnected network with non-negligible probability.

• *Concave shape 2:* We skip results for the first concave shape in figure 2(c) because they are similar to those of the U shaped obstacle and turn to the final obstacle. As seen in figure 2(d), this obstacle is problematic. The message is routed out of the obstacle only to fall back in with high probability. As a consequence, the random version of GRIC only reaches acceptable performances for very high network densities: success rate is bad for densities below 5 and path length is prohibitive even for densities below 8, c.f. figures 3(g) and 3(h).

4 Conclusion

We have studied geographic routing in the presence of hard communication blocking obstacles and proposed a new way of routing messages which substantially improves the state of the art by somehow combining the best of two worlds: the lightweight (no topology maintenance overhead), robustness (to link failure) and simplicity of the greedy routing algorithm with the high success rates and obstacle avoidance features of face routing. The simplicity of GRIC suggests that it would be a protocol of choice for routing in mobile networks. We shall investigate this in future work. We have shown that GRIC resists (and actually performs better) in the presence of limited link failure. Future work will investigate this matter more in depth, as well as the question of localization errors. At first sight, there seems to be no reason to believe GRIC to be sensitive to them. GRIC proposes an alternative to the face family of protocols. We believe it has a slightly different application niche and is preferable in the case of highly dynamic networks (because frequent topology changes increase the topology maintenance overhead of the planarization phase required for face routing), whereas face routing may be better in sparse but stable topologies where some overhead is acceptable. Deeper understanding of the differences between face and GRIC routing will require further investigation, more realistic communication graph models and possibly turning to real world experiments.

References

1. Rabaey, J.M., Josie Ammer, M., da Silva, J.L., Patel, D., Roundy , S.: Picoradio supports ad hoc ultra-low power wireless networking. Computer (2000)
2. Warneke, B., Last, M., Liebowitz, B., Pister, K.S.J.: Smart dust: communicating with a cubic-millimeter computer. Computer (2001)
3. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks (2002)
4. Haenggi, M.: Challenges in wireless sensor networks. In: Handbook of Sensor Networks: Compact Wireless and Wired Systems, CRC Press, Boca Raton (2005)
5. Estrin, D., Govindan, R., Heidemann, J., Kumar, S.: Next century challenges: Scalable coordination in sensor networks. In: Mobile Computing and Networking, ACM Press, New York (1999)
6. Karl, H., Willig, A.: Protocols and Architectures for Wireless Sensor Networks. Wiley, Chichester (2005)
7. Zhao, F., Guibas, L.: Wireless Sensor Networks, an Information Processing Approach. Elsevier, Amsterdam (2004)
8. Finn, G.G.: Routing and addressing problems in large metropolitan-scale internetworks. Technical report, Information Sciences Institute (1987)
9. Seada, K., Helmy, A., Govindan, R.: On the effect of localization errors on geographic face routing in sensor networks. In: Information Processing in Sensor Networks (2004)
10. Beutel, J.: Location management in wireless sensor networks. In: Handbook of Sensor Networks: Compact Wireless and Wired Systems, CRC Press, Boca Raton (2005)
11. Hightower, J., Borriello, G.: Location systems for ubiquitous computing. Computer (2001)
12. Leone, P., Moraru, L., Powell, O., Rolim, J.: A localization algorithm for wireless ad-hoc sensor networks with traffic overhead minimization by emission inhibition. In: Algorithmic Aspects of Wireless Sensor Networks (2006)
13. Rao, A., Ratnasamy, S., Papadimitriou, C., Shenker, S., Stoica, I.: Geographic routing without location information. In: Mobile Computing and Networking (2003)
14. Ahmed, N., Kanhere, S.S., Jha, S.: The holes problem in wireless sensor networks: a survey. SIGMOBILE Mob. Comput. Commun. Rev. (2005)
15. Bose, P., Morin, P., Stojmenovic, I., Urrutia, J.: Routing with guaranteed delivery in ad hoc wireless networks. In: Discrete Algorithms and Methods for Mobile Computing and Communications (1999)
16. Karp, B., Kung, H.T.: GPSR: greedy perimeter stateless routing for wireless networks. In: Mobile Computing and Networking (2000)
17. Kuhn, F., Wattenhofer, R., Zollinger, A.: Worst-case optimal and average-case efficient geometric ad-hoc routing. In: Mobile ad hoc Networking & Computing (2003)
18. Powell, O., Nikoletseas, S.: Geographic routing around obstacles in wireless sensor networks. Technical report, Computing Research Repository (CoRR) (2007)
19. Kim, Y.-J., Govindan, R., Karp, B., Shenker, S.: On the pitfalls of geographic face routing. In: Foundations of mobile computing (2005)
20. Kim, Y.-J., Govindan, R., Karp, B., Shenker, S.: Geographic routing made practical. In: Networked Systems Design & Implementation (2005)
21. Kim, Y.-J., Govindan, R., Karp, B., Shenker, S.: Lazy cross-link removal for geographic routing. In: Embedded Networked Sensor Systems (2006)

22. Fang, Q., Gao, J., Guibas, L.: Locating and bypassing holes in sensor networks. Mobile Networks and Applications (2006)
23. Chatzigiannakis, I., Dimitriou, T., Nikoletseas, S., Spirakis, P.: A probabilistic forwarding protocol for efficient data propagation in sensor networks. Journal of Ad hoc Networks (2006)
24. Antoniou, T., Chatzigiannakis, I., Mylonas, G., Nikoletseas, S., Boukerche, A.: A new energy efficient and fault-tolerant protocol for data propagation in smart dust networks using varying transmission range. In: Annual Simulation Symposium (ANSS). ACM/IEEE (2004)
25. Chatzigiannakis, I., Nikoletseas, S., Spirakis, P.: Smart dust protocols for local detection and propagation. Journal of Mobile Networks (MONET) (2005)
26. Chatzigiannakis, I., Kinalis, A., Nikoletseas, S.: Efficient and robust data dissemination using limited extra network knowledge. In: Gibbons, P.B., Abdelzaher, T., Aspnes, J., Rao, R. (eds.) DCOSS 2006. LNCS, vol. 4026, Springer, Heidelberg (2006)
27. Chatzigiannakis, I., Mylonas, G., Nikoletseas, S.: Modeling and evaluation of the effect of obstacles on the performance of wireless sensor networks. In: Annual Simulation Symposium (ANSS). ACM/IEEE (2006)
28. Hemmerling, A.: Labyrinth Problems: Labyrinth-Searching Abilities of Automata. B.G. Teubner, Leipzig (1989)
29. Yu, Y., Hong, B., Prasanna, V.: On communication models for algorithm design for networked sensor systems: A case study. Pervasive and Mobile Computing (2005)
30. Cerpa, A., Wong, J., Kuang, L., Potkonjak, M., Estrin, D.: Statistical model of lossy links in wireless sensor networks. In: Information processing in sensor networks (2005)

A Distributed Primal-Dual Heuristic for Steiner Problems in Networks

Marcelo Santos[1], Lúcia M.A. Drummond[1], and Eduardo Uchoa[2]

[1] Department of Computer Science, Fluminense Federal University, Brazil
{msantos,lucia}@ic.uff.br
[2] Department of Production Engineering, Fluminense Federal University, Brazil
uchoa@producao.uff.br

Abstract. Multicast routing problems are often modeled as Steiner Problems in undirected or directed graphs, the later case being particularly suitable to cases where most of the traffic has a single source. Sequential Steiner heuristics are not convenient in that context, since one cannot assume that a central node has complete information about the topology and the state of a large wide area network. This paper introduces a distributed version of a primal-dual heuristic (known as Dual Ascent), known for its remarkable good practical results, lower and upper bounds, in both undirected and directed Steiner problems. Experimental results and complexity analysis are also presented, showing the efficiency of the proposed algorithm when compared with the best distributed algorithms in the literature.

1 Introduction

The Steiner Problem in Graphs (SPG) is defined as follows. Given an undirected graph $G = (V, E)$, positive edge costs c and a set $T \subseteq V$ of *terminal nodes*, find a connected subgraph (V', E') of G with $T \subseteq V'$ minimizing $\sum_{e \in E'} c_e$. In other words, find a minimum cost tree containing all terminals, possibly also containing some non-terminal nodes. The Steiner Problem in Directed Graphs (SPDG) is the case where $G_D = (V, A)$ is a directed graph and there is a special root terminal $r \in T$. The problem is to find a minimum cost directed tree containing paths from r to every other terminal. Both the SPG and the SPDG are NP-hard, one must resort to heuristic algorithms if solutions must be obtained in short time.

Several emerging network applications, like teleconferencing or video on demand, require the transmission of large amounts of data among a small subset of the nodes. This is called *multicast* or *selective broadcast*, the usual broadcast being the case where the information must be sent to all nodes in the network. The routing of multicast connections is the problem of establishing message paths for a multicast session. Such routing problem is often modelled as a SPG, as surveyed by Novak et al. [7] and also by Oliveira and Pardalos [9]. The frequent situation where most multicast messages have a single source and the network

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 175–188, 2007.
© Springer-Verlag Berlin Heidelberg 2007

is asymmetric, i.e., link characteristics like latency, capacity, congestion or price depend on the direction, is better modelled as a SPDG.

Sequential Steiner heuristics are not much suitable for multicast routing, since one cannot assume that a central node has complete information about the topology and the state of a large wide area network. The overhead to collect, store and update this information could be prohibitive. In this context, there is a need for distributed algorithms, where each node initially only knows about its immediate neighborhood.

A simple approach to this distributed problem is constructing a MST and remove subtrees without terminals [2], but this usually leads to poor solutions. More sophisticated algorithms in the literature are distributed versions of the Shortest Path Heuristics (SPH). The Prim-SPH (a.k.a. Cheapest Insertion Heuristic) grows a single tree, starting with a chosen terminal, called the root. At each step a least cost path is added, from the existing partially built tree to a terminal not yet connected. Its distributed versions [1,13] construct, in parallel, shortest paths from each node to each non-root terminal. Those shortest paths are used by another parallel thread, that starts from the root to build the Steiner Tree. The time complexity of those algorithms, measured by the maximum sequence of messages, is $O(|T|.|V|)$. The overall number of exchanged messages is $O(|V|^2)$. Novak et al. [8] proposed improvements on those algorithms leading to a better practical performance, but could not change the worst case complexities. The above mentioned distributed Prim-SPH can be adapted to the SPDG.

The so-called Kruskal-SPH (although it actually resembles Borüvka's MST algorithm) grows several subtrees at once, starting at each terminal. At each step some pairs of subtrees are joined by shortest paths. Its distributed version was proposed by Bauer and Varma [1], with complexities $O(|T|.|V|)$ time and $O(|V|^2)$ messages. This last complexity was improved to $O(|V| \log |V|)$ by Singh and Vellanki [15]; this also improves the time complexity when $|T|$ is not $O(\log |V|)$. Those algorithms (or even the sequential Kruskal-SPH) cannot be adapted to the SPDG.

The Average Distance Heuristic (ADH) also starts with subtrees composed by each terminal. At each step a pair of subtrees is joined by a path passing by the non-terminal node with minimum average distance to each subtree. The distributed version by Gatani et al. [3] takes $O(|T|.|V|)$ time and $O(|E|+|T|.|V|)$ messages. The ADH cannot be adapted to the SPDG.

Those distributed algorithms (Prim-SPH, Kruskal-SPH and ADH) assume that each node already knows its shortest distance to all other nodes. If this is not the case, the distributed computation of such distances would add a message complexity of $O(|E|.|V|)$, a bit complexity of $O(|E|.|V|. \log |V|)$ and a time complexity of $O(|V|)$ [14].

The SPG and SPDG algorithm proposed in this article is a distributed version of the sequential dual ascent algorithm proposed by Wong [18]. This algorithm has the following advantages over other heuristics.

- Extensive computational experiments over the main classes of SPG benchmark instances from the literature have shown that Dual Ascent usually yields better solutions [16,10,17,12,11] than Prim-SPH. Kruskal-SPH and ADH are a little worse than Prim-SPH.
- The Dual Ascent is an example of what was later called a *primal-dual algorithm* [4], it can be interpreted as working in the dual of a linear program formulation. In practice, this means that Dual Ascent not only returns a solution, it also returns a guarantee of the quality of this solution. Dual Ascent lower bounds are remarkably good, they are usually less than 3% below the optimal. For this reason, it is a key part of the best exact algorithms for the SPG [12,10]. Tight lower bounds can be very useful. If the user is not satisfied with the guarantee obtained, he may want to run the Dual Ascent again (this distributed algorithm is not deterministic), or any other heuristic, trying to get better solutions. Moreover, the lower bounds can be used to remove arcs from the instance, by proving that they do not belong to an optimal solution. It is typical to remove more than half of the arcs. A second run of Dual Ascent (or of any good heuristic) on that reduced instance is quite likely to improve the solution.
- It does not require that nodes know the value of least cost paths to every other node. Some authors [1,8,3] argue that since network layer protocols like RIP and OSPF already keep routing tables with that information, it could be used by their Steiner algorithms. Such reasoning is not completely satisfactory since it relies on particular technologies that may be supplanted in the future. Moreover, using network layer information limits what can be accomplished by the multicast protocol. RIP routing tables actually provides hop distances, i.e., least cost paths assuming that all links have unitary costs. OSPF routing tables use the link costs provided by local administrators (usually as a function of parameters like latency time or available bandwidth). RIP or OSPF costs are not necessarily the more appropriate for building multicast trees. This application may prefer using its own link costs, reflecting factors like contractual prices or forecasted link behavior after the multicast begins. Of course, one can always run a shortest distance algorithm with respect to the desired costs before computing the tree. However, this would add the above mentioned complexities [14] to the overall method.

The proposed distributed Dual Ascent has worst case complexities of $O(|V|^2)$ global time, $O(|T|.|V|^2)$ messages and $O(|V|)$ local time complexity. One alternative to perform the Dual Ascent would be electing a leader node to locally solve the problem and then broadcast the solution. It would take $O(|V|.|E|)$ messages and $O(|V|)$ time to concentrate all the relevant information about G in the leader, the local time complexity of the sequential Dual Ascent algorithm is $O(|E|^2)$. Considering such complexities and the memory demand at the leader node, the fully distributed approach is an appealing alternative for multicast routing. A recent example of the distribution of another primal-dual algorithm is given by Grandoni et al.[5].

The remainder of this paper is organized as follows. Section 2 introduces the sequential Wong's Dual Ascent Algorithm. The distributed algorithm is presented and analyzed in Section 3. Section 4 presents experimental results, comparing the practical performance of our algorithm with Prim-SPH in terms of solution quality, time and exchanged messages.

2 Sequential Wong's Dual Ascent Algorithm

Each arc a has its non-negative *reduced cost* $\bar{c}_a$, a value that is initialized with the original arc cost. Reduced costs may be only decreased. Arcs with zero reduced cost are called *saturated*. Those arcs induce the *saturation graph* $G_S = (V, A_r(\bar{c}))$, where $A_r(\bar{c}) = \{a \in A : \bar{c}_a = 0\}$. As all original costs are positive, this graph starts with no arcs. All operations in the algorithm are defined over the current saturation graph. Let R be the set of nodes of a strongly connected component of G_S, i.e., a maximal set containing a directed cycle passing by any pair of nodes in it. This set is a *root component* if (i) R contains a terminal, (ii) R does not contains r, and (iii) there is no terminal $t \notin R$ reaching R by a path in G_S. Given a root component R, define $W(R) \supseteq R$ as the set of nodes reaching R by paths in G_S and let $\delta^-(W)$ be the directed cut consisting of the arcs entering W. The DA algorithm follows:

Wong's Dual Ascent

$LB \leftarrow 0$; $\bar{c}_a \leftarrow c_a$, for all $a \in A$;
While (exists root components in G_S) {
    Choose a root component R; $W \leftarrow W(R)$;
    $\Delta \leftarrow \min_{a \in \delta^-(W)} \bar{c}_a$;
    $\bar{c}_a \leftarrow \bar{c}_a - \Delta$, for all $a \in \delta^-(W)$;
    $LB \leftarrow LB + \Delta$;
}
Output: $A_r(\bar{c})$ and LB

At first, each terminal other than the root corresponds to a root component. In each round, a root component R is chosen and the reduced costs of all arcs incident to $W(R)$ are decreased by Δ, the smallest such reduced cost. The partial lower bound is increased by the same amount. At least one arc is saturated in each round. Some saturations reduce the number of root components, until there are no root components anymore. At this point, G_S contains at least one directed path from r to every other terminal. Therefore it contains at least one solution.

3 The Distributed Algorithm

3.1 General Idea

We propose an asynchronous distributed version of the previously described algorithm. Its output is the lower bound and the arc reduced costs, which also

gives the saturation graph. The cleaning steps to actually compute a solution from this graph can be performed by already known distributed algorithms. We assume that each node knows the cost of each arc incident to that node. During the execution, all data about the state of an arc is kept by its adjacent nodes. Each node performs the same local algorithm, which consists of sending messages over adjoining links, waiting for incoming messages, and processing. Messages can be transmitted independently and arrive after an unpredictable but finite delay, without error and in sequence. We view the nodes in the graph as being initially asleep. All non-root terminals wake up spontaneously, other nodes awake upon receiving messages from awakened neighbors. We assume that the nodes of the graph have distinct identities that can be totally ordered.

The agents in this distributed algorithm are associated to the *fragments*, defined as a set $W(t)$ formed by all nodes reaching a non-root terminal t. The terminal t identifying a fragment $W(t)$ is also known as its *leader*. Note that if t belongs to a strongly connected component R, $W(t) = W(R)$. Strongly connected components are disjoint by definition, but sets $W(R)$ are not. Therefore, a node can work on several fragments, perhaps belonging to different connected components.

By definition, all nodes in a fragment are connected to its leader by saturated arcs. Among such arcs, the algorithm keeps in each fragment a tree used for intra-fragment message exchange, *convergecast messages* from nodes in the fragment to the leader and *broadcast messages* from leader to nodes. A fragment *growing round* consists of finding the minimum reduced cost of an incident arc and subtracting this value from the reduced cost of all incident arcs. The first operation is performed using a convergecast, the result is then broadcast. Since the decrease of reduced costs causes some new arcs to become saturated, the corresponding nodes must be included in the fragment and the fragment tree updated. The fragment leader keeps the partial lower bound due to the fragment growing rounds. As fragments grow, their nodes may start to overlap. Having common incident arcs creates a mutual exclusion problem, that causes some fragments to suspend their operations temporarily. A fragment should stop its execution when it is reached from the root. When the root reaches all terminals, it initiates a broadcast to terminate the algorithm.

3.2 Growing Fragments

At first, a fragment composed by just a terminal t selects the incident arc with the minimum reduced cost, at this point the original cost. The fragment partial lower bound is increased from zero to this value, which is subtracted from the reduced cost of each incident arc. Messages *Include(t)* are transmitted on the saturated arcs. The node sending the *Include(t)* messages, marks those arcs as *To_leaf(t)*. Those arcs define a directed tree from the leader to all other nodes in the fragment, for broadcast operations. Upon receiving this message, a node includes itself in the fragment and marks the outgoing arc to the node that sent the message as *To_leader(t)*. The arcs marked as *To_leader(t)* define a directed tree pointing to the leader, used in convergecast operations. This node also send

Check(t) messages to all its other neighbors, communicating that it now belongs to t and asking if they are also part of t. Those messages must be answered, *Ack(t,y)* if the neighbor also belongs to t and *Ack(t,n)* if it does not.

The next step is calculating the minimum reduced cost of all arcs incident to the fragment. A leaf node in the fragment sends by its *To_leader(t)* arc a message *Conv(t,Smallest,Δ)* with the minimum reduced cost of an arc incident to it, Δ, and not belonging to the fragment. The internal nodes in the fragment, upon receiving its *Conv(t,Smallest,Δ)* messages, compare those costs with the minimum reduced cost of an arc incident to it and not belonging to the fragment. The smallest such value is sent by its *To_leader(t)* arc using another *Conv(t,Smallest,Δ)* message. When the fragment leader finally knows the fragment minimum reduced cost, it updates the fragment lower bound and starts broadcasting this value back to all fragment nodes, using *Broad(t,Smallest,Δ)* messages. Upon receiving such messages, a node decreases the cost of its incident arcs, which may trigger *Include(t)* messages on the newly saturated arcs, therefore growing the fragment. The newly added nodes first checks its neighborhood before starting another round of convergecast. Leaf nodes that did not include any new node start a new round of convergecast immediately. See Figure 1, for an example.

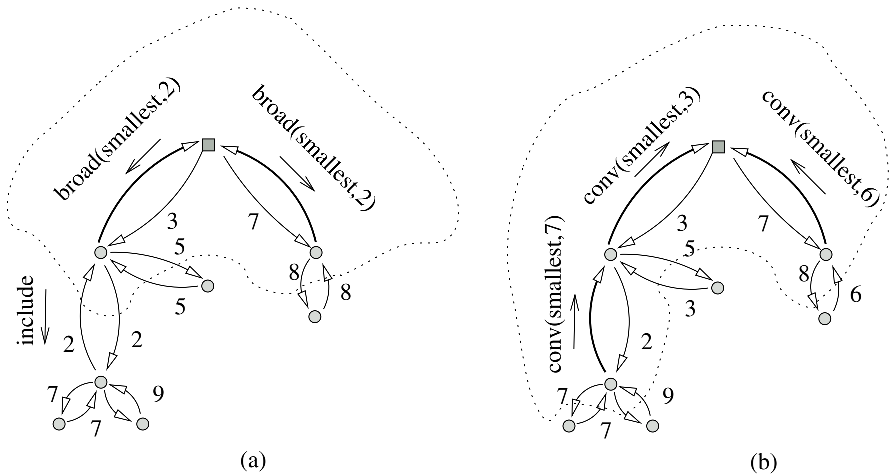


Fig. 1. Growing Fragments

When several nodes are included in a fragment in the same growing round, it is possible that a newly added node i receives a *Ack(t,n)* from a node j that will still receive a *Include(t)* in that round. This means that i will consider arc (j, i) as incident to the fragment, it is possible that its reduced cost reaches the leader as the smallest. When a *Broad(t,Smallest,Δ)* with this value reaches i, this node will be already informed (by a *Check(t)* message) that j is actually

part of the fragment. Arc (j, i) will be saturated but no *Include(t)* message will be sent by it. Therefore, it is possible to have *degenerated growing rounds*, i.e., rounds where only arcs internal to the fragment are saturated and no new node is included. However, it is not possible to have two degenerated growing rounds in sequence.

Another situation happens when nodes i and j send *Include(t)* messages to the same node k. Suppose that the message from i arrives first. The second *Include(t)* message should be answered by a *AlreadyIncluded(t)* message. Node j then knows that (j, k) should not be marked as *To_leaf(t)*.

Finally, it is also possible that a newly included node also includes other nodes in the same growing round. Suppose that node i is included. It sends *Check(t)* messages and waits for the corresponding *Ack(t)* messages. Then it gets the minimum reduced cost of an incident arc not belonging t. If this value is positive, i knows that it is a fragment leaf and sends a *Conv(t,Smallest,Δ)* message as usual. However, since other fragments are also decreasing reduced costs, it is possible that this value is zero, i.e., there are saturated arcs (i, j) such that j does not belongs to t. In this case, i sends *Include(t)* messages to nodes j, that will continue the growing round.

3.3 Suspending Fragments

Fragments with common incidents arc cannot grow at the same time, since the reduced cost of those arcs would be changed concurrently. Here we have a classical mutual exclusion problem, where the shared resource is the common incident arcs. In order to solve this conflict, only the fragment with the largest identification will go on growing, while the other is suspended. It remains suspended until the fragment in conflict finishes its execution or is suspended by another fragment. When a node that already belongs to some fragments receives an include message from another fragment, there is a potential conflict. In order to simplify the algorithm, without sacrificing its correctness and complexities, this condition is enough to suspend all but one of the fragments. At a given moment, each node is associated to at most one active fragment.

Two cases should be considered. When a node receives *Include*(t_1) from a fragment with identification smaller than its current active fragment t_2, it answers *Conv*(t_1,*Suspend*) immediately. On the other case, when the identification of t_1 is greater than t_2, the node must wait until t_2 finishes its current growing round, before suspending t_2 and continue the growing of t_1.

The leader of a suspended fragment t_2 sends *Broad*(t_2,*Suspend*) messages, indicating to all its nodes that it was suspended. When a conflict node receives such a message indicating that its current active fragment t_2 was suspended, it tries to reactivate other fragments, like t_1, that were suspended by it. This is done by sending to their leaders a *Conv*(t_1,*Reactivate*) message. In case other conflicts do not exist, the leader of t_1 will send *Broad(t,Reactivate)* messages to re-initiate its growing. The overall situation is illustrated in Figure 2. Fragments t_1 and t_2 have a common node j, t_1 is suspended. Then t_2 grows and conflicts

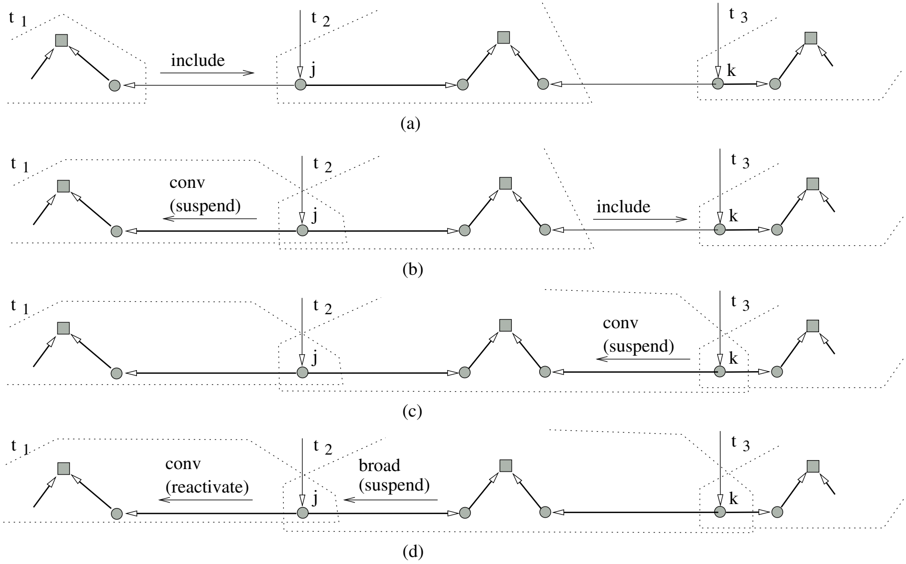


Fig. 2. Suspending and Reactivating Fragments

with t_3 at node k, t_2 is suspended and re-activates t_1. When a fragment finishes, as will be described in the next section, similar reactivation messages may be also sent.

3.4 Stopping Fragments

A fragment may stop growing definitively because it can be reached from the root node by saturated arcs. When the root node receives a *Include(t)*, it sends back a *Conv(t,End,root)*. The fragment leader then sends a *Broad(t,End,root,lb)* with its partial lower bound and stops. The root node accumulates those values to obtain the global lower bound. When a fragment is stopped it also tries to reactivate other suspended fragments in conflict with it, as occurs when a fragment is suspended.

A conflict may also occur involving a shared terminal. It happens when a fragment identified by t_1 sends a *Include*(t_1) to another active fragment leader t_2, connecting t_2 to t_1. The same two cases, previously described, should also be considered. When a node receives *Include*(t_1) from a fragment with identification smaller than its current active fragment t_2, it immediately answers *Conv*(t_1,*Suspend*). When fragment t_2 is reached by the root, it can reactivate fragment t_1 that in this case will also be reached by the root in its next growing iteration. On the other case, when the identification of t_1 is greater than t_2, t_2 is suspended and t_1 continues its growing. In case of fragment t_1 includes the root, it stops growing definitively and it tries to reactivate fragment t_2. If the root belongs to the subtree rooted in t_2, in its next growing iteration, fragment

t_2 will also be able to stop growing. In case of t_1 is suspended, fragment t_2 could be reactivated but if the conflict that suspended t_1 belongs to the subtree rooted in t_2, it will remain suspended. When the root reaches all terminals, it initiates a broadcast to terminate the algorithm.

3.5 Algorithm Analysis

Communication Cost: We want an upper bound on the number of messages exchanged during the execution of the algorithm. The worst case would occur in a complete graph where the terminals (including the root) are connected by high cost arcs, in such a way they will only be saturated when all non-terminal nodes are already included in all fragments. When a node is included in a fragment, it checks all its $|V|$ neighbors, this will be done $|V|-|T|$ times for each of the $|T|-1$ fragments. Therefore, there can be up to $O(|T|.|V|^2)$ *Check* and *Ack* messages. We now show that no other message exceeds that bound. Each growing round demands $O(|V|)$ *Conv*, *Broad* and *Include* messages. As there can be up to $O(|V|)$ growing rounds by fragment, $O(|T|.|V|^2)$ is a valid upper bound on the number of such messages.

Considering the suspending and reactivation procedures, the worst situation occurs when we have $|T|$ fragments whose leaders, are $t_1, t_2, \ldots, t_{|T|}$, such that t_1 conflicts with t_2 at a node, t_2 conflicts with t_3 at another node and so on. Assume those leaders are ordered in increasingly identification order. Then, t_1 is suspended while t_2 goes on executing. When t_3 suspends t_2, t_1 is reactivated. Then t_4 suspends t_3, t_2 is reactivated and t_1 is suspended again. The total of reactivations is $O(|T|^2)$. For each reactivation, a broadcast on the tree is executed, that will never require more than $O(|V|)$ messages.

Time Cost: Intuitively, as more active fragments execute, more arcs are saturated in parallel. Although it appears that the algorithm allows an amount of parallelism limited to the number of terminals, there are particular situations where there is almost no parallelism. As in the previous section, the worst case would occur in a complete graph where the terminals (including the root) are connected by high cost arcs, in such a way they will only be saturated when all non-terminal nodes are already included in all fragments.

Let us suppose that we have a fragment where one arc is saturated at each broadcast, and that when it stops, the resulting tree has two subtrees, one that is a unique long chain of saturated arcs and the other containing the root. For each new node included in this fragment, the causal chain of message receiving and sending is increased permanently by two, corresponding to the receiving and the sending of *Broad* and *Conv* messages in the next growing rounds. At the moment a new node is included a causal chain two long, corresponding to the sending of *Check* and the receiving of *Ack* messages, is also formed, but this chain occurs only in the round the node is included. So the resulting causal chain considering broadcasts, convergecasts and the checks is $O(V^2)$. Remark that the same may occur for only one more fragment. After this, a connected component is formed allowing for all other fragments, in a broadcast wave that includes any node

of such component, the incorporation of all other nodes of the component in a linear time. Summarizing, for only two fragments the causal chain is $O(V^2)$, the others will be reached by the root in $O(V)$ time. So, the complexity is $O(V^2)$.

4 Experimental Results

Our distributed algorithm was implemented in C and MPICH2-1.0.3 and executed on a cluster with 15 Athlon 1.8 GHz processors. Half of the tests were performed over the SPG benchmark instances from the SteinLib mentioned in Table 1. Instances from B series are random graphs with different densities and random edge costs between 1 and 10. Instances from I080 series are also random graphs with different densities, but edges costs were chosen to make them hard to solve. Instances from P4Z series have complete graphs. All instances in Steinlib are undirected. Since the proposed algorithm was designed for the more general case of digraphs, we created SPDG instances from the above mentioned SPG instances to perform the remaining tests. Each edge in the original graph is replaced by a pair of opposite arcs. Their costs are the original costs multiplied by a random factor uniformly distributed in the range [0.5,1.5] and rounded. A random terminal is chosen to be the root. We also implemented the distributed version of the Prim-SPH found in [7] (adapted to digraphs), including a shortest distance algorithm. This Prim-SPH was applied as a stand-alone algorithm and also to find the solution contained in the reduced graphs produced by our distributed DA algorithm.

Columns in table 1 have the following meaning: $|V|$, $|E|$, and $|T|$ give the instance size; **Opt** is the value of the optimal solution (calculated with the branch-and-bound code from [10]); **SPHc** is the value of the solution obtained by running Prim-SPH as a stand-alone algorithm; **LB** is the lower bound given by Dual Ascent; next column gives the proportion of the arcs that are saturated; **DA+SPHr** is the value of the solution obtained running the Prim-SPH over the graph containing only the saturated arcs. Remark that the results given for the Dual Ascent are averages of 5 runs. Since processor load and communication times may change on each execution, the sequence of fragment growing in the DA may also change, leading to slightly different results. The average standard deviation on the results of all executions is only 0.4%.

We use *competitiveness*, the ratio of the heuristic cost and the optimal cost, to compare the quality of the solutions provided by DA and Prim-SPH. Charts in figures 3 and 4 show the cumulative percentage of cases whose competitiveness is less than or equal a given value, for undirected and directed instances. It is clear that DA + SPHr gives better solutions than SPHc. We also charted the improved results of executing both DA + SPHr and SPHc and taking the best solution. However, there is a more interesting approach to obtain similar results. Using the lower bounds provided by DA, we can evaluate the solution obtained with DA + SPHr and execute SPHc only if it exceeds a given limit. We applied this idea, executing SPHc only if (DA + SPHr) / LB exceeded 1.05, this happened

Table 1. Instances used on experiences with some results

Instance				Undirected					Directed				
Name	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	Opt	SPHc	LB	$\frac{\lvert Ar\rvert}{\lvert A\rvert}$%	DA+SPHr	Opt	SPHc	LB	$\frac{\lvert Ar\rvert}{\lvert A\rvert}$%	DA+SPHr
b01	50	63	9	82	85	82.0	13.9	**84.6**	77	82	77.0	13.9	**77.0**
b02	50	63	13	83	84	82.4	13.5	**83.8**	101	107	100.4	13.4	**104.0**
b03	50	63	25	138	**138**	136.4	12.6	139.2	170	176	165.4	10.3	**173.4**
b04	50	100	9	59	63.6	58.8	13.6	**59.6**	58	61	57.4	13.5	**59.6**
b05	50	100	13	61	**62**	60.8	13.6	**62.0**	61	**64**	60.6	13.6	65.4
b06	50	100	25	122	**127**	121.6	13.5	132.2	128	134	126.8	13.2	**129.6**
b07	75	94	13	111	**111**	110.8	13.7	112.0	122	**122**	121.6	13.6	125.0
b08	75	94	19	104	**104**	104.0	13.8	106.0	115	126	114.8	13.7	**116.0**
b09	75	94	38	220	**220**	216.0	11.7	221.2	240	244	239.2	13.4	**242.4**
b10	75	150	13	86	91	86.0	13.8	**89.8**	90	**91**	89.0	13.5	92.6
b11	75	150	19	88	**90**	87.8	13.7	90.8	103	**104**	100.8	13.1	105.2
b12	75	150	38	174	**174**	174.0	13.8	175.0	168	**171**	161.0	11.5	174.0
b13	100	125	17	165	**174**	155.2	9.5	180.0	176	202	163.8	13.3	**179.4**
b14	100	125	25	235	**238**	230.6	10.0	238.4	250	261	234.0	13.4	**250.4**
b15	100	125	50	318	**318**	317.6	13.6	320.4	342	356	341.0	13.4	**345.6**
b16	100	200	17	127	136	124.6	13.1	**132.4**	133	**133**	131.6	13.4	141.6
b17	100	200	25	131	**132**	128.2	13.0	133.0	139	**142**	137.0	13.3	145.2
b18	100	200	50	218	225	215.0	13.0	**218.0**	258	271	258.0	13.8	**259.8**
i080-001	80	120	6	1787	2164	1770.6	19.8	**1787.0**	1751	1815	1729.4	16.3	**1780.8**
i080-011	80	350	6	1479	1671	1462.4	22.3	**1596.0**	1220	1296	1220.0	17.4	**1227.6**
i080-021	80	3160	6	1175	**1471**	1159.8	16.0	1476.0	741	847	673.0	5.8	**768.4**
i080-031	80	160	6	1570	**1570**	1570.0	17.0	**1570.0**	1514	1706	1455.0	16.5	**1552.8**
i080-041	80	632	6	1276	1600	1192.8	19.1	**1279.0**	946	1026	900.0	5.9	**983.0**
i080-101	80	120	8	2608	3009	2608.0	17.3	**2772.0**	2322	2706	2322.0	10.5	**2322.0**
i080-111	80	350	8	2051	**2142**	1792.6	15.5	2262.8	1580	1600	1499.2	13.2	**1580.2**
i080-121	80	3160	8	1561	2054	1539.4	17.0	**1844.0**	977	1097	910.0	5.3	**1050.0**
i080-131	80	160	8	2284	2561	2238.0	19.5	**2555.6**	1875	2061	1844.0	17.9	**2016.0**
i080-141	80	632	8	1788	2058	1662.4	22.9	**1865.2**	1404	1596	1265.6	10.1	**1404.0**
i080-201	80	120	16	4760	5435	4555.2	20.6	**5129.0**	4321	4502	4228.4	22.7	**4322.8**
i080-211	80	350	16	3631	4132	3259.6	16.7	**3839.0**	2951	3174	2808.6	12.8	**3141.0**
i080-221	80	3160	16	3158	4386	3063.6	14.6	**3499.0**	1985	2293	1799.0	5.1	**2082.0**
i080-231	80	160	16	4354	**4834**	3961.0	22.5	4917.0	4156	4623	4036.0	14.0	**4415.2**
i080-241	80	632	16	3538	4463	3247.6	20.3	**3805.0**	2492	**2617**	2351.2	8.7	2746.8
i080-301	80	120	20	5519	**5628**	5223.4	15.2	5630.0	5714	6365	5390.8	24.8	**5916.0**
i080-311	80	350	20	4554	5853	4215.6	19.2	**5115.8**	3471	**3813**	3266.6	12.9	4065.0
i080-321	80	3160	20	3932	5527	3925.0	11.1	**4461.0**	2453	3137	2374.0	4.1	**2755.4**
i080-331	80	160	20	5226	5947	4831.2	21.6	**5589.0**	4506	4911	4258.6	19.3	**4877.0**
i080-341	80	632	20	4236	5728	3997.2	17.3	**4529.0**	3132	**3361**	3025.0	15.4	3412.0
P401	100	4950	5	155	170	149.8	0.4	**157.0**	145	165	145.0	0.4	**145.0**
P402	100	4950	5	116	**116**	116.0	0.2	**116.0**	102	**102**	102.0	0.2	**102.0**
P403	100	4950	5	179	**184**	176.0	0.5	199.0	169	186	166.0	0.5	**180.0**
P404	100	4950	10	270	**270**	243.2	0.5	**270.0**	270	**279**	214.8	0.3	285.0
P405	100	4950	10	270	291	268.6	0.3	**270.0**	248	250	243.2	0.5	**248.0**
P406	100	4950	10	290	319	286.0	0.5	**290.0**	281	303	226.8	0.3	**296.0**
P407	100	4950	20	590	**601**	576.8	0.3	609.0	546	590	523.0	0.4	**575.0**
P408	100	4950	20	542	559	520.4	0.3	**551.0**	502	520	480.2	0.3	**502.0**

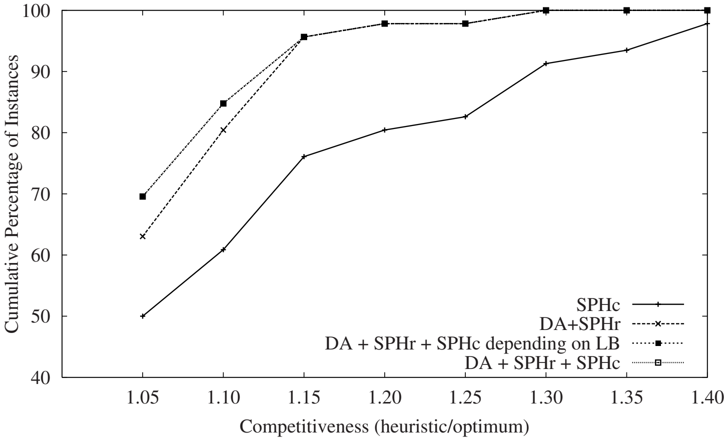


Fig. 3. Solution quality – undirected instances

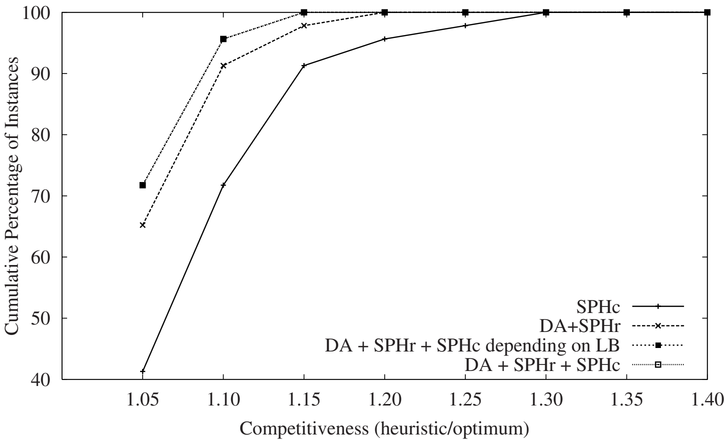


Fig. 4. Solution quality – directed instances

in 49% of the cases. Those results are also charted, in fact, the last two curves are undistinguishable.

We also compare the practical performance of DA and Prim-SPH performance with respect to time, given as the size of the largest message sequence, and to the total number of exchanged messages. Those measurements, for each directed instance, grouped by series, are shown in figures 5 and 6. For the sake of space, we omit similar measurements on undirected instances. It can be seen that DA is indeed more costly than Prim-SPH on most instances, taking more time and exchanging more messages. However, it was never much more costly than Prim-SPH, their performance differences were always within a small factor

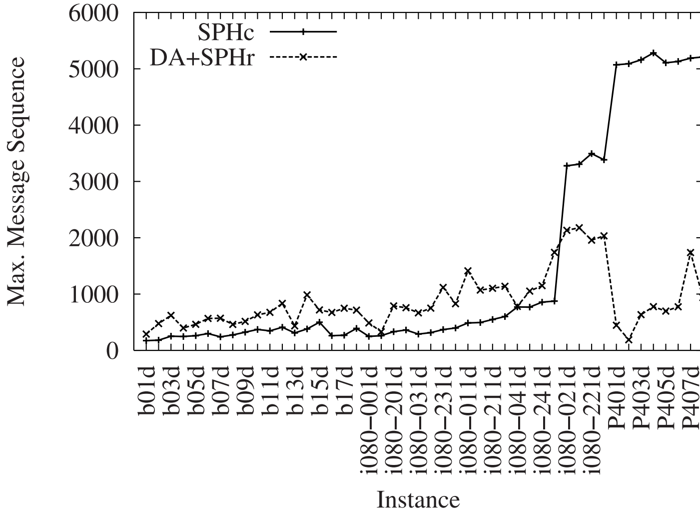


Fig. 5. Time – directed instances

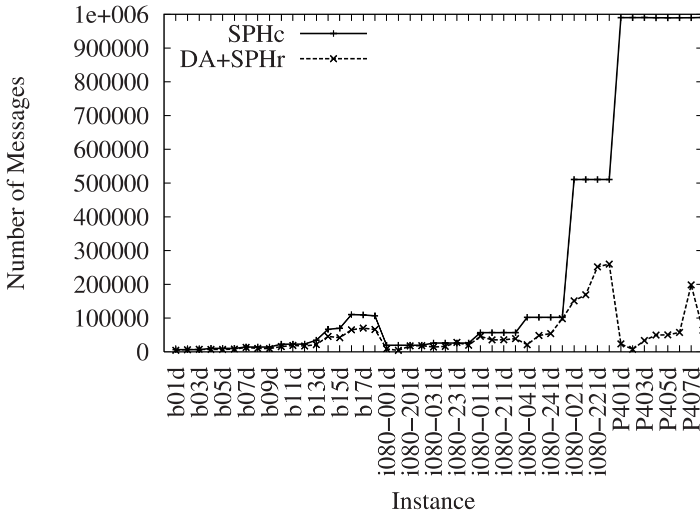


Fig. 6. Messages – directed instances

of 4. Moreover, DA performed better on larger I080 instances and much better on P4Z instances. This happens because the cost of running Prim-SPH in those instances is dominated by the calculation of shortest distances among all nodes of large graphs.

References

1. Bauer, F., Varma, A.: Distributed algorithms for multicast path setup in data networks. IEEE/ACM Transactions on Networking 4, 181–191 (1996)
2. Chen, G., Houle, M., Kuo, M.: The Steiner problem in distributed computing systems. Information Sciences 74, 73–96 (1993)
3. Gatani, L., Re, G.L.: Gaglio, S.: An efficient distributed algorithm for generating multicast distribution trees. In: Proc. of the 34th ICPP - Workshop on Performance Evaluation of Networks for Parallel, Cluster and Grid Computer Systems, pp. 477–484 (2005)
4. Goemans, M., Williamson, D.: The primal-dual method for approximation algorithms and its application to network design. In: Approximation algorithms for NP-hard problems, Dorit S. Hochbaum (ed), PWS Publishing Co., 144–191 (1996)
5. Grandoni, F., Könemann, J., Panconesi, A., Sozio, M.: Primal-dual based distributed algorithms for vertex cover with semi-hard capacities. In: Proc. of the 24th ACM SIGACT-SIGOPS pp. 839–848 (2005)
6. Koch, T., Martin, A., Voss, S.: SteinLib: an updated library on Steiner Problems in Graphs, ZIB-Report 00-37 http://elib.zib.de/steinlib (2000)
7. Novak, R., Rugelj, J., Kundus, G.: Steiner tree based distributed multicast routing in networks. In: Steiner Trees in Industries, pp. 327–352. Kluwer, Dordrecht (2001)
8. Novak, R., Rugelj, J., Kundus, G.: A note on distributed multicast routing in point-to-point networks. Computers and Operations Research 26, 1149–1164 (2001)
9. Oliveira, C., Pardalos, P.: A survey of combinatorial optimization problems in multicast routing, Computers and Operations Research, 32, 1953–1981 (2005)
10. de Aragão, M.P., Uchoa, E., Werneck, R.: Dual heuristics on the exact solution of Large Steiner Problems. ENDM 7, 46–51 (2001)
11. de Aragão, M.P., Werneck, R.: On the implementation of MST-based heuristics for the Steiner problem in graphs. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 1–15. Springer, Heidelberg (2002)
12. Polzin, T., Vahdati, S.: Improved algorithms for the Steiner problem in networks. Discrete Applied Mathematics 112, 263–300 (2001)
13. Rugelj, J., Klavzar, S.: Distributed multicast routing in point-to-point networks. Computers and Operations Research 24, 521–527 (1997)
14. Segall, A.: Distributed network protocols. IEEE Trans. on Information Theory 29, 23–35 (1983)
15. Singh, G., Vellanki, K.: A distributed protocol for constructing multicast trees. In: Proc. of the 2nd International Conference on Principles of Distributes Systems pp. 61–76 (1998)
16. Voss, S.: Steiner's problem in graphs: heuristic methods. Discrete Applied Mathematics 40, 45–72 (1992)
17. Werneck, R.: Steiner Problem in Graphs: primal, dual, and exact algorithms, Master's Thesis, Catholic University of Rio de Janeiro (2001)
18. Wong, R.: A dual ascent approach for Steiner Tree Problems on a directed graph. Mathematical Programming 28, 271–287 (1984)

An Experimental Study of Stability in Heterogeneous Networks

Maria Chroni[1], Dimitrios Koukopoulos[2], and Stavros D. Nikolopoulos[1]

[1] Department of Computer Science, University of Ioannina,
GR-45110 Ioannina, Greece
{mchroni,stavros}@cs.uoi.gr
[2] Department of Applied Informatics in Management and Finance,
Technological Education Institute of Ionian Islands,
and University of Ioannina, GR-30100 Agrinio, Greece
koukopou@ceid.upatras.gr

Abstract. A distinguishing feature of today's large-scale communication networks, such as the Internet, is their *heterogeneity,* predominantly manifested by the fact that a wide variety of *communication protocols* are simultaneously running over different network hosts. A fundamental question that naturally poses itself for such common settings of heterogeneous networks concerns their ability to preserve the number of packets in the system upper bounded at all times. This property is well-known as stability. We focus on the *Adversarial Queueing Theory* framework, where an adversary controls the rates of packet injections and determines packet paths. In this work, we present specific network constructions with different protocol compositions and we show experimentally their stability behavior under an adversarilly strategy. In particular, we study compositions of universally stable protocols with unstable protocols like FIFO. Interestingly, some of our results indicate that such a composition leads to a worst stability behavior than having a single unstable protocol for contention-resolution. This suggests that the potential for instability incurred by the composition of *one* universally stable protocol with *one* unstable protocol may be *worse* than that of some *single* protocol.

1 Introduction

Motivation-Framework. *Objectives.* Recent years, a lot of research has been done in the field of packet-switched communication networks for the specification of their behavior. A major issue that arises in such networks is that of *stability*– will the number of packets in the network remain bounded at all times? The answer to this question may depend on the *rate* of injecting packets into the network, and the composition of *protocols* running on different network hosts in order to resolve packet conflicts. In this work, we embark on a study of the impact of heterogeneity of packet-switched networks on their performance properties. We choose, as a test-bed, the case of distinct *communication protocols* that are simultaneously running on different hosts in a distributed system. We

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 189–202, 2007.
© Springer-Verlag Berlin Heidelberg 2007

ask, in particular, which (and how) stability properties of greedy, contention-resolution protocols operating over a *packet-switched* communication network are maintained under composition of such protocols.

Framework of Adversarial Queueing Theory. We consider a packet-switched network in which packets arrive dynamically at the nodes with predetermined paths, and they are routed at discrete time steps across the edges. We focus on a basic adversarial model for packet arrival and path determination that has been introduced by Borodin *et al.* [4], under the name *Adversarial Queueing Theory.* Roughly speaking, this model views the time evolution of a packet-switched communication network as a game between an *adversary* and a *protocol*. At each time step, the adversary may inject a set of packets into some nodes with a specified simple path. When more than one packets wish to cross a queue at a given time step, a *contention-resolution* protocol is employed to resolve the conflict. A crucial parameter of the adversary is its *injection rate* $\rho \in (0, 1)$.

Stability. Roughly speaking, a protocol P is *stable* [4] on a network $\mathcal{G}$ against an adversary $\mathcal{A}$ of rate ρ if there is a constant B (which may depend on $\mathcal{G}$ and $\mathcal{A}$) such that the number of packets in the system is bounded at all times by B. On the other hand, *a protocol* P *is universally stable* [4] if it is stable against every adversary of rate less than 1 and on every network. Till our work, the study of stability has focused on *homogeneous* networks, that is, on networks in which the same contention-resolution protocol is running at all queues or *heterogeneous* networks, in which compositions of universally stable contention-resolution protocols are running at all queues. In this work, we embark on a study of the effect of composing unstable contention-resolution protocols with universally stable protocols on the stability of the resulting system. (By *composition* of contention-resolution protocols, we mean the simultaneous use of different such protocols at different queues of the system.)

In particular, we study the stability properties of three unstable protocols when they are composed with stable protocols. The unstable protocols are: *First-In-First-Out* (FIFO), which gives priority to the packet that has arrived first in the queue, *Nearest-to-Go* (NTG), which gives priority to the packet whose distance to its destination is minimal and *Furthest-From-Source* (FFS), which advances the packet whose distance to its destination is maximal. The stable protocols are: *Farthest-to-Go* (FTG), which gives priority to the packet whose distance to its destination is maximal *Nearest-to-Source* (NTS), which gives priority to the packet whose distance traversed is minimal, *Longest-in-System* (LIS), which gives priority to the packet injected the earliest, and *Shortest-in-System* (SIS), which gives priority to the packet most recently injected.

Contribution. In this work, we initiate the study of the stability properties of heterogeneous networks with compositions of universally-stable and unstable, greedy, contention-resolution protocols, such as LIS, SIS, FTG, NTS, FFS, NTG and FIFO, running on top of them [5]. In particular, we design three different network constructions, and we apply three different adversarial strategies, one for

each of the networks. Then, we experimentally evaluate the stability properties of the networks on various scenarios of protocol compositions. Our results are three-fold; they are summarized as follows:

- We establish that, the composition of FIFO with any of the universally-stable protocols SIS, LIS, NTS and FTG is not universally-stable. Surprisingly, the composition of FIFO with any of SIS and NTS protocols in some network constructions results in lower bounds on injection rate for network instability comparing to the single usage of FIFO into the same networks.
- We establish that, the composition of NTG with any of the universally-stable protocols SIS, LIS, NTS and FTG is not universally-stable.
- We establish that, the composition of FFS with any of the universally-stable protocols SIS and NTS is not universally-stable.

We feel that our study of the instability properties of networks hosting compositions of universally-stable and unstable contention-resolution protocols (within the context of Adversarial Queueing Theory) provides an important step towards understanding the impact of heterogeneity on the performance properties of large-scale communication networks such as the Internet.

Related Work. The issue of composing distributed protocols (resp., objects) to obtain other protocols (resp., objects), and the properties of the resulting (*composed*) protocols (resp., objects), has a rich record in Distributed Computing Theory (see, e.g., [12]). For example, Herlihy and Wing [7] establish that a composition of *linearizable* memory objects (possibly distinct), each managed by its own protocols, preserves linearizability. Adversarial Queueing Theory [4] received a lot of interest in the study of stability and instability issues (see, e.g., [1,2,6,10]). The universal stability of various natural greedy protocols (LIS, SIS, NTS and FTG) was established by Andrews *et al.* [2]. Also, FFS and NTG have been proved unstable for injection rate $\rho > 1/\sqrt{2}$ [2]. The instability of FIFO (in the model of adversarial queueing theory) was first established by Andrews *et al.* [2, Theorem 2.10] for injection rate $\rho \geq 0.85$ (for the network $\mathcal{G}_1$ of Figure 1a). Lower bounds of 0.8357 (for the network $\mathcal{G}_2$ of Figure 1b) and 0.749 (for the network $\mathcal{G}_3$ of Figure 1c) on FIFO instability were presented by Diaz *et al.* [6, Theorem 3] and Koukopoulos *et al.* [8, Theorem 5.1]. An alternative approach for studying protocol instability in the context of adversarial queueing theory is based on parameterized constructions for networks with *unbounded size*. Using this approach, Bhattacharjee and Goel [3] show that FIFO can become unstable for arbitrarily small injection rates. Moreover, Tsaparas [13] based on the same approach proved that NTG is unstable at arbitrarily low rates of injection. The subfield of study of the stability properties of compositions of protocols was introduced by Koukopoulos *et al.* in [8,9,10] where lower bounds of 0.683, 0.519 and 0.5 on the injection rates that guarantee instability for the composition pairs LIS-SIS, LIS-NTS and LIS-FTG were presented.

2 Preliminaries

The model definitions are patterned after those in [4, Section 3]. We consider that a routing network is modelled by a directed graph $\mathcal{G} = (V, E)$. Each node $u \in V$ represents a communication switch, and each edge $e \in E$ represents a link between two switches. In each node, there is a buffer (queue) associated with each outgoing link. Time proceeds in discrete time steps. Buffers store packets that are injected into the network with a route, which is a simple directed path in $\mathcal{G}$. A *packet* is an atomic entity that resides at a buffer at the end of any step. It must travel along paths in the network from its *source* to its *destination,* both of which are nodes in the network. When a packet is injected, it is placed in the buffer of the first link on its route. When a packet reaches its destination, we say that it is *absorbed*. During each step, a packet may be sent from its current node along one of the outgoing edges from that node.

Any packets that wish to travel along an edge e at a particular time step, but are not sent, wait in a queue for e. At each step, an *adversary* generates a set of requests. A *request* is a *path* specifying the route that will be followed by a packet.[1] We say that the adversary generates a set of packets when it generates a set of requested paths. Also, we say that a packet p *requires* an edge e at time t if e lies on the path from its position to its destination at time t.

The definition of a *bounded adversary* A of rate (ρ, b) (where $b \geq 1$ is a natural number and $0 < \rho < 1$) [4] requires that for any edge e and any interval I, the adversary injects no more than $\rho|I| + b$ packets during I that require edge e at their time of injection. Such a model allows for adversarial injection of packets that are "bursty" using the integer $b > 0$.

When we consider adversarial constructions for proving instability of compositions of specific protocols in which we want to derive lower bounds, it is advantageous to have an adversary that is as weak as possible. Thus, for these purposes, we say that an adversary A has injection rate ρ if for every $t \geq 1$, every interval I of t steps, and every edge e, it injects no more than $\rho|t|$ packets during I that require edge e at the time of their injection.

In order to formalize the behavior of a network, we use the notions of *system* and *system configuration.* A triple of the form $\langle \mathcal{G}, \mathcal{A}, \mathsf{P} \rangle$ where $\mathcal{G}$ is a network, $\mathcal{A}$ is an adversary and P is the used protocol (or list of protocols) on the network queues is called a system. In every time step t, the current configuration C^t of a system $\langle \mathcal{G}, \mathcal{A}, \mathsf{P} \rangle$ is a collection of sets $\{S_e^t : e \epsilon \mathcal{G}\}$, such that S_e^t is the set of packets waiting in the queue of the edge e at the end of step t.

3 Unstable Compositions of Protocols

In this section we focus on networks that already have been proved to be unstable under FIFO protocol. We describe the adversary's strategy for each network and we present the stability results for various protocol compositions.

[1] In this work, it is assumed, as it is common in packet routing, that all paths are simple paths where edges cannot be overlapped, while vertices can be overlapped.

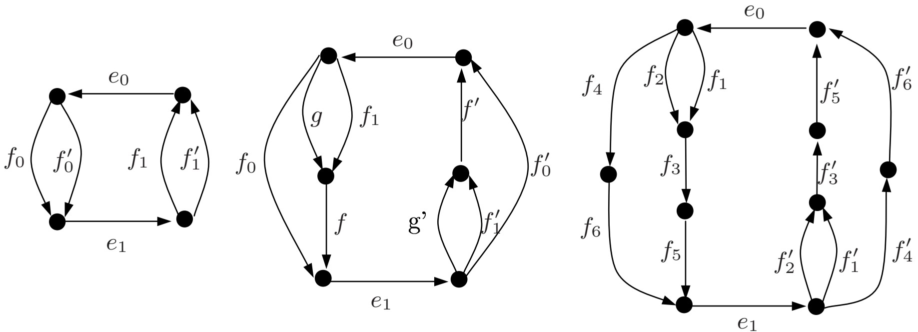


Fig. 1. Network Constructions $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$

3.1 Stability Behavior of $\mathcal{G}_1$ Network

Adversary's strategy. Consider the network $\mathcal{G}_1$ in Figure 1. Time is split into phases. At the beginning of phase j, there are s_j packets that are queued in the queue e_0 requiring to traverse the edge e_0. We will construct an adversary $\mathcal{A}_1$ such that at the beginning of phase $j+1$ there will be s_{j+1} packets that will be queued in the queue e_1, requiring to traverse the edge e_1. During phase j the adversary plays three rounds of injections as follows:

Round 1: For the first s_j time steps, the adversary injects in e_0 a set X of ρs_j packets that want to traverse the edges e_0, f'_0, e_1.

Round 2: For the next ρs_j time steps, the adversary injects in e_0 a set Y of $\rho^2 s_j$ packets that want to traverse the edges e_0, f_0, e_1 and in f'_0 a set Z of $\rho^2 s_j$ packets that want to traverse the edge f'_0. The adversary uses Z packets to delay X packets in f'_0.

Round 3: For the next $\rho^2 s_j$ time steps, the adversary injects in e_1 $\rho^3 s_j$ packets that want to traverse the edge e_1.

At the end of Round 3, there will be a number of packets (s_{j+1}) that will be queued in the queue e_1, requiring to traverse the edge e_1. Then, during phase $j+1$ the same adversarial construction is applied for the symmetric edges using the quantity s_{j+1} instead of s_j. In order to guarantee instability, it suffices to show that $s_{j+1} > s_j$ for each successive phases j and $j+1$.

Instability of Protocol Compositions for $\mathcal{G}_1$: We apply the adversary $\mathcal{A}_1$ on network $\mathcal{G}_1$ for the compositions of protocols FIFO, NTG, FFS with any of FTG, NTS, SIS, LIS. Table 1 shows the instability properties of such compositions.

In order to show that our experimental analysis agrees with theoretical analysis, we prove:

Theorem 1. Let $\rho \geq 0.76$. There is a network $\mathcal{G}_1$ and an adversary $\mathcal{A}_1$ of rate ρ, such that the system ($\mathcal{G}_1$, $\mathcal{A}_1$, FIFO, NTS) is unstable.

Table 1. Network $\mathcal{G}_1$: Instability of FIFO, NTG, FFS with any of FTG, NTS, SIS, LIS

Protocol Compositions	Unstable?	Injection rate ρ
FIFO - FTG	YES	$\rho \geq 0.86$
FIFO - NTS	YES	$\rho \geq 0.76$
FIFO - SIS	YES	$\rho \geq 0.76$
FIFO - LIS	YES	$\rho \geq 0.86$
NTG - FTG	YES	$\rho \geq 0.76$
NTG - NTS	YES	$\rho \geq 0.76$
NTG - SIS	YES	$\rho \geq 0.77$
NTG - LIS	YES	$\rho \geq 0.77$
FFS - FTG	NO [2]	-
FFS - NTS	YES	$\rho \geq 0.77$
FFS - SIS	YES	$\rho \geq 0.77$
FFS - LIS	NO	-

Proof. Consider the network $\mathcal{G}_1$ in Figure 1. All the queues use FIFO to resolve packet conflicts except from queues f_0 and f_1 that use NTS.

Inductive Hypothesis: At the beginning of phase j, there are s_j packets that are queued in the queue e_0 requiring to traverse the edge e_0.

Induction Step: At the beginning of phase $j+1$ there will be more than s_j packets that will be queued in the queue e_1, requiring to traverse the edge e_1.

We will construct an adversary A such that the induction step will hold. Proving that the induction step holds, we ensure that the inductive hypothesis will hold at the beginning of phase $j+1$ for the symmetric edges with an increased value of s_j, $s_{j+1} > s_j$. In order to prove that the induction step works, we should consider that there is a large enough number of packets s_j in the initial system configuration. During phase j the adversary plays three rounds of injections. The sequence of injections is as follows:

Round 1: For the first s_j time steps, the adversary injects a set X of ρs_j packets that want to traverse the edges $e_0, f_0^{'}, e_1$. These packets are blocked by the initial s_j packets, because queue e_0 uses FIFO protocol.

Round 2: For the next ρs_j time steps, the adversary injects a set Y of $\rho^2 s_j$ packets that want to traverse the edges e_0, f_0, e_1. At the same time, the adversary inserts a set Z of $\rho^2 s_j$ which will traverse queue $f_0^{'}$. During this round, all the packets of set Z, will traverse $f_0^{'}$ because these packets are nearest to their source. Moreover, $\rho s_j - \rho^2 s_j)$ packets of set X traverse queue f_0, so at the end of this round there are $\rho^2 s_j$ packets of set X in queue f_0.

Round 3: For the next $\rho^2 s_j$ time steps, the adversary injects a set W of $\rho^3 s_j$ packets that want to traverse edge e_1. During this round, in queue e_1 arrive the $\rho^2 s_j$ packets of set Y, and the $\rho^2 s_j$ packets of set X. So, at the end of this round, there are in queue e_1 $\rho^2 s_j + \rho^3 s_j$ packets. For $\rho \geq 0.76$, it holds that $s_{j+1} = \rho^2 s_j + \rho^3 s_j > s_j$. □

3.2 Stability Behavior of $\mathcal{G}_2$ Network

Adversary's strategy. Consider the network $\mathcal{G}_2$ in Figure 1. Time is split into phases. At the beginning of phase j, there are s_j packets that are queued in the queue e_0 requiring to traverse the edges e_0, g, f. We will construct an adversary $\mathcal{A}_2$ such that at the beginning of phase $j+1$ there will be s_{j+1} packets that will be queued in the queue e_1, requiring to traverse the edges e_1, g', f'. During phase j the adversary plays three rounds of injections as follows:

Round 1: During the first s_j time steps, the adversary injects into e_0 a set X of ρs_j packets requiring to traverse the edges e_0, f_1, f, e_1, g', f'. Also, the adversary injects into queue g a set Y of ρs_j packets targeted to g.

Round 2: During the next ρs_j time steps, the adversary injects into e_0 a set Z of $\rho^2 s_j$ packets requiring to traverse e_0, f_0, e_1, g', f'. Also, the adversary injects into f a set W of $\rho^2 s_j$ packets with destination f.

Round 3: During the next $\left(\rho + \frac{1}{1+\rho}\right)$ time steps, the adversary injects into e_1 a set V of packets requiring to traverse the edges e_1, g', f'.

At the end of Round 3, there will be a number of packets (s_{j+1}) that will be queued in the queue e_1, requiring to traverse the edges e_1, g', f'. Then, during phase $j+1$ the same adversarial construction is applied for the symmetric edges using the quantity s_{j+1} instead of s_j. In order to guarantee instability, it suffices to show that $s_{j+1} > s_j$ for each successive phases j and $j+1$.

Instability of Protocol Compositions for $\mathcal{G}_2$: We apply the adversary $\mathcal{A}_2$ on network $\mathcal{G}_2$ for the compositions of contention-resolution protocols FIFO, NTG, FFS with any of FTG, NTS, SIS, LIS. Table 2 shows the instability properties of such compositions.

3.3 Stability Behavior of $\mathcal{G}_3$ Network

Adversary's strategy. Consider the network $\mathcal{G}_3$ in Figure 1. Time is split into phases. At the beginning of phase j, there are s_j packets (in total) that are queued in the queues $e_0, f_3', f_4', f_5', f_6'$ requiring to traverse the edges e_0, f_1, f_3, f_5, all these packets manage to depart their initial edges to the symmetric part of the network (f_1, f_3, f_5), as a continuous flow in s_j time steps, and the number of packets that are queued in queues f_4', f_6' is bigger than the number of packets queued in queues f_3', f_5'. We will construct an adversary $\mathcal{A}_3$ such that at the beginning of phase $j+1$ there will be s_{j+1} packets queued in the queues f_3, f_5, f_4, f_6, e_1 requiring to traverse e_1, f_1', f_3', f_5', all of which will be able to depart their initial edges to the symmetric part of the network (f_1', f_3', f_5') in s_{j+1} time steps as a continuous flow and the number of packets that are queued in queues f_4, f_6 is bigger than the number of packets queued in queues f_3, f_5. During phase j the adversary plays three rounds of injections as follows:

Table 2. Network $\mathcal{G}_2$: Instability of FIFO, NTG, FFS with any of FTG, NTS, SIS, LIS

Protocol Compositions	Unstable?	Injection rate ρ
FIFO - FTG	YES	$\rho \geq 0.84$
FIFO - NTS	YES	$\rho \geq 0.84$
FIFO - SIS	YES	$\rho \geq 0.84$
FIFO - LIS	YES	$\rho \geq 0.84$
NTG - FTG	NO	-
NTG - NTS	NO	-
NTG - SIS	NO	-
NTG - LIS	NO	-
FFS - FTG	NO	-
FFS - NTS	YES	$\rho \geq 0.73$
FFS - SIS	YES	$\rho \geq 0.73$
FFS - LIS	NO	-

Round 1: For s_j time steps, the adversary injects in $f_4^{'}$ a set X of ρs_j packets wanting to traverse edges $f_4^{'}, f_6^{'}, e_0, f_2, f_3, f_5, e_1, f_1^{'}, f_3^{'}, f_5^{'}$. At the same time, the adversary injects in queue f_1 a set S_1 of ρs_j packets.

Round 2: For the next ρs_j time steps, the adversary injects a set Y of $\rho^2 s_j$ packets requiring $f_4^{'}, f_6^{'}, e_0, f_4, f_6, e_1, f_1^{'}, f_3^{'}, f_5^{'}$. At the same time, the adversary injects a set S_2 of $\rho^2 s_j$ packets in f_2, a set S_3 of $\rho^2 s_j$ packets in f_3, and a set S_4 of $\rho^2 s_j$ packets in f_5.

Round 3: For the next $\rho^2 s_j$ time steps, the adversary injects a set S_5 of $\rho^3 s_j$ packets requiring to traverse f_4. Furthermore, the adversary injects a set Z of $\rho^3 s_j$ packets requiring to traverse $f_6, e_1, f_1^{'}, f_3^{'}, f_5^{'}$.

At the end of Round 3, there will be a number of packets (s_{j+1}) that will be queued in the queues f_3, f_5, f_4, f_6, e_1 requiring to traverse $e_1, f_1^{'}, f_3^{'}, f_5^{'}$. Then, during phase $j+1$ the same adversarial construction is applied for the symmetric edges using the quantity s_{j+1} instead of s_j. In order to guarantee instability, it suffices to show that $s_{j+1} > s_j$ for each successive phases j and $j+1$.

Instability of Protocol Compositions for $\mathcal{G}_3$: We apply the adversary $\mathcal{A}_3$ on network $\mathcal{G}_3$ for the compositions of contention-resolution protocols FIFO, NTG, FFS with any of FTG, NTS, SIS, LIS. Table 3 shows the instability properties of such compositions.

4 Experimental Evaluation

In order to evaluate the stability properties of various protocol compositions we carried an experimental study. All of our implementations follow closely the

Table 3. Network $\mathcal{G}_3$: Instability of FIFO, NTG, FFS with any of FTG, NTS, SIS, LIS

Protocol Compositions	Unstable?	Injection rate ρ
FIFO - FTG	YES	$\rho \geq 0.76$
FIFO - NTS	YES	$\rho \geq 0.73$
FIFO - SIS	YES	$\rho \geq 0.73$
FIFO - LIS	YES	$\rho \geq 0.76$
NTG - FTG	NO	-
NTG - NTS	NO	-
NTG - SIS	NO	-
NTG - LIS	YES	$\rho \geq 0.8$
FFS - FTG	NO	-
FFS - NTS	NO	-
FFS - SIS	NO	-
FFS - LIS	NO	-

network constructions, the adversarial strategies and the properties of contention-resolution protocols we described above. They have been implemented as C++ classes by using C++ Builder.

The simulation environment that we developed is based on the Adversarial Queueing Model presented in Section 2 and allows to perform an experiment for given symmetric or non-symmetric network constructions, a fixed packet injection rate, a given adversarial strategy, a fixed contention-resolution protocol or a composition of protocols used on different network queues, a specified number of phases and a given amount of initial packets in the network along with their placement into the network queues.

The experiments were conducted on a Windows box (Windows XP, Pentium III at 933MHz, with 512MB memory at 133MHz) using C++ Builder.

We experimented on compositions of universally stable and unstable protocols. In particular, we are interested in the behavior of the number of packets of all the network queues in successive phases for various compositions of protocols. If the total number of packets in the network queues increases at any times, then the network is unstable. Figures 2, 3, and 4 illustrate our experiments considering the worst injection rate we estimated with respect to instability for any composition of protocols we studied.

The results of our experiments are summarized in the tables below; we state only the results that lead to instability. The information of which queues use which protocol is included into the following tables. For example, in Table 4, in the cell that corresponds to the composition FIFO - NTS the line $[f_0, f_1](\rho \geq 0.76)$ means that all the queues use FIFO except of f_0, f_1 which use NTS protocol and the injection rate lower bound that guarantees instability is $\rho \geq 0.76$.

Generally, we formulated our experiments in two categories: in the first category all queues use the same unstable protocol except of one that uses a

Table 4. Instability of protocol compositions on network $\mathcal{G}_1$

	NTS	SIS	FTG	LIS
FIFO	$[f_0, f_1]$ $(\rho \geq 0.76)$ $[f_0]$ $(\rho \geq 0.85)$ $[f'_0, f_1]$ $(\rho \geq 0.85)$ $[f'_0]$ $(\rho \geq 0.86)$ $[f'_0, f'_1]$ $(\rho \geq 0.86)$	$[f_0, f_1]$ $(\rho \geq 0.76)$ $[f_0]$, $(\rho \geq 0.85)$ $[f'_0, f_1]$ $(\rho \geq 0.85)$ $[f'_0]$ $(\rho \geq 0.86)$ $[f'_0, f'_1]$ $(\rho \geq 0.86)$	$[f'_0]$ $(\rho \geq 0.86)$ $[f'_0, f'_1]$ $(\rho \geq 0.86)$	$[e_0]$ $(\rho \geq 0.86)$ $[e_1, f'_1]$ $(\rho \geq 0.86)$
NTG	$[f_0]$ $(\rho \geq 0.76)$ $[f'_0, f'_1]$ $(\rho \geq 0.76)$ $[f_1]$ $(\rho \geq 0.77)$	$[f_0]$ $(\rho \geq 0.77)$ $[f'_0, f'_1]$ $(\rho \geq 0.77)$	$[f'_0]$ $(\rho \geq 0.76)$ $[f'_0, f'_1]$ $(\rho \geq 0.76)$	$[f'_0]$ $(\rho \geq 0.77)$ $[f'_0, f'_1]$ $(\rho \geq 0.77)$
FFS	$[f_0, f_1]$ $(\rho \geq 0.77)$	$[f_0, f_1]$ $(\rho \geq 0.77)$	NO	NO

Table 5. Instability of protocol compositions on network $\mathcal{G}_2$

	NTS	SIS	FTG	LIS
FIFO	$[f_1]$ $(\rho \geq 0.84)$ $[f_1, f'_1]$ $(\rho \geq 0.84)$ $[g]$ $(\rho \geq 0.87)$ $[g, g']$ $(\rho \geq 0.9)$	$[f_1]$ $(\rho \geq 0.84)$ $[g]$ $(\rho \geq 0.87)$ $[g, g']$ $(\rho \geq 0.9)$	$[f_1]$ $(\rho \geq 0.84)$ $[f_1, f'_1]$ $(\rho \geq 0.84)$ $[g]$ $(\rho \geq 0.87)$ $[g, g']$ $(\rho \geq 0.87)$	$[f_1]$ $(\rho \geq 0.84)$ $[f_1, f'_1]$ $(\rho \geq 0.84)$ $[g]$ $(\rho \geq 0.87)$ $[g, g']$ $(\rho \geq 0.87)$
NTG	NO	NO	NO	NO
FFS	$[g, g']$ $(\rho \geq 0.73)$ $[f, f']$ $(\rho \geq 0.79)$	$[g, g']$ $(\rho \geq 0.73)$ $[f, f']$ $(\rho \geq 0.79)$	NO	NO

Table 6. Instability of protocol compositions on network $\mathcal{G}_3$

	NTS	SIS	FTG	LIS
FIFO	$[f_2, f'_2]$ $(\rho \geq 0.73)$ $[f_6]$ $(\rho \geq 0.76)$ $[f_6, f'_6]$ $(\rho \geq 0.76)$	$[f_2, f'_2]$ $(\rho \geq 0.73)$ $[f_5, f'_5]$ $(\rho \geq 0.76)$	$[f_6]$ $(\rho \geq 0.76)$ $[f_5]$ $(\rho \geq 0.76)$ $[f_5, f'_5]$ $(\rho \geq 0.76)$	$[f_4, f'_4]$ $(r \geq 0.76)$
NTG	NO	NO	NO	NO
FFS	NO	NO	NO	NO

universally stable protocol, while in the second category all queues use the same unstable protocol except of two that both use the same universally stable protocol. In both of the two categories of experiments we assumed that initially there are s_0=1000 packets in the system. In addition, all of the experiments are executed for 80 phases.

We start our experimentation by considering the effect of the composition of FIFO with NTS, FTG, LIS, SIS protocols on the stability properties of network $\mathcal{G}_1$. Figure 2b depicts the total number of packets into the queues of $\mathcal{G}_1$ under the compositions of FIFO with any of NTS, FTG, LIS, SIS protocols. Furthermore, for comparison reasons, we estimate the evolution of the number of packets into the network when FIFO is used for contention-resolution on all queues of $\mathcal{G}_1$ (Figure 2a). Finally, Figures 2c and 2d depict the total number of packets into

the queues of $\mathcal{G}_1$ under the compositions of NTG and FFS protocols with any of NTS, FTG, LIS, SIS protocols.

The results of the experiments on network $\mathcal{G}_1$ (Table 4) show that the composition of an unstable protocol with a universally stable protocol is unstable for the most composition pairs. Surprisingly, in the case of the composition pairs FIFO-NTS, and FIFO-SIS we found a lower bound on the injection rate ($\rho \geq 0.76$) that guarantees instability than the instability lower bound specified in [2] ($\rho \geq 0.85$) applying only FIFO on $\mathcal{G}_1$.

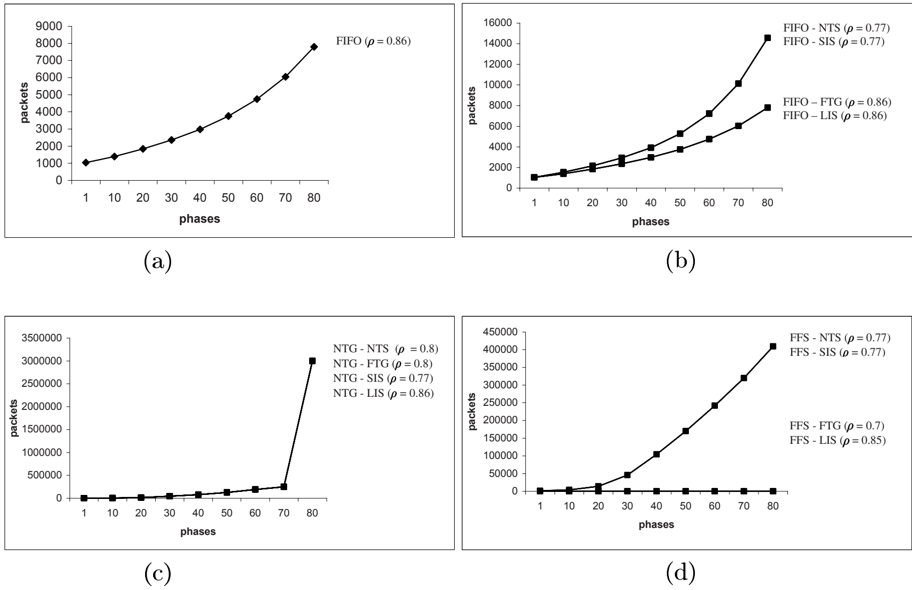


Fig. 2. Instability curves of $\mathcal{G}_1$ under a protocol or a composition of protocols: (a) FIFO, (b) FIFO-NTS, FIFO-SIS, FIFO-FTG, FIFO-LIS, (c) NTG-NTS, NTG-SIS, NTG-FTG, NTG-LIS, (d) FFS-NTS, FFS-SIS, FFS-FTG, FFS-LIS

After $\mathcal{G}_1$, we study the effect of composing unstable protocols FIFO, NTG, FFS with universally-stable protocols NTS, FTG, LIS, SIS on network $\mathcal{G}_2$. Figure 3b depicts the total number of packets into the queues of $\mathcal{G}_2$ under the compositions of FIFO with any of NTS, FTG, LIS, SIS protocols. Furthermore, for comparison reasons, we estimate the evolution of the number of packets into the network when FIFO is used for contention-resolution on all queues of $\mathcal{G}_2$ (Figure 3a). Finally, Figures 3c and 3d depict the total number of packets into the queues of $\mathcal{G}_2$ under the compositions of NTG and FFS protocols with any of NTS, FTG, LIS, SIS protocols.

The results of the experiments on network $\mathcal{G}_2$ (Table 5) show that the instability property of FIFO is maintained, even though we compose it with universally stable protocols. On the other hand, the NTG protocol loses its instability property when it is composed with any of universally stable protocols FTG, NTS, SIS

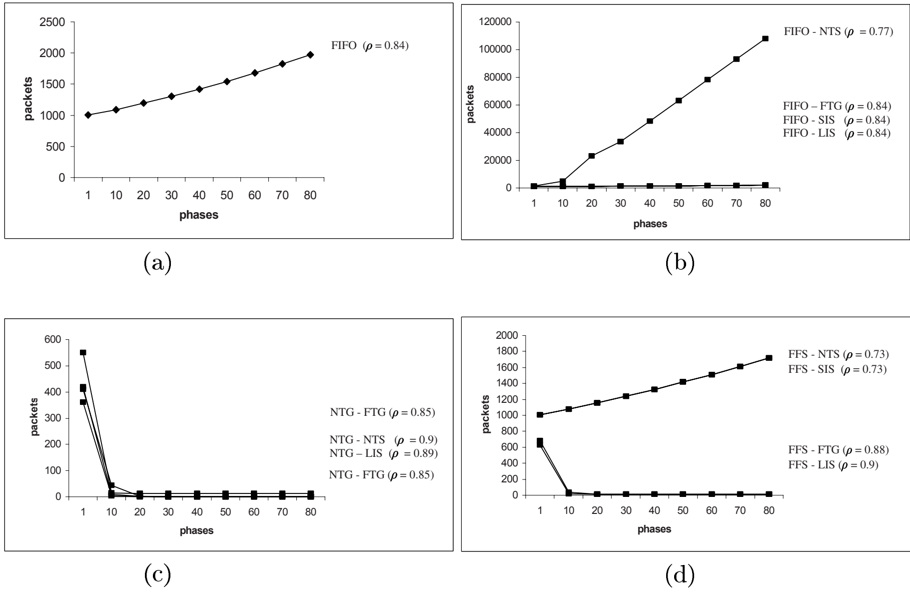


Fig. 3. Instability curves of $\mathcal{G}_2$ under a protocol or a composition of protocols: (a) FIFO, (b) FIFO-NTS, FIFO-SIS, FIFO-FTG, FIFO-LIS, (c) NTG-NTS, NTG-SIS, NTG-FTG, NTG-LIS, (d) FFS-NTS, FFS-SIS, FFS-FTG, FFS-LIS

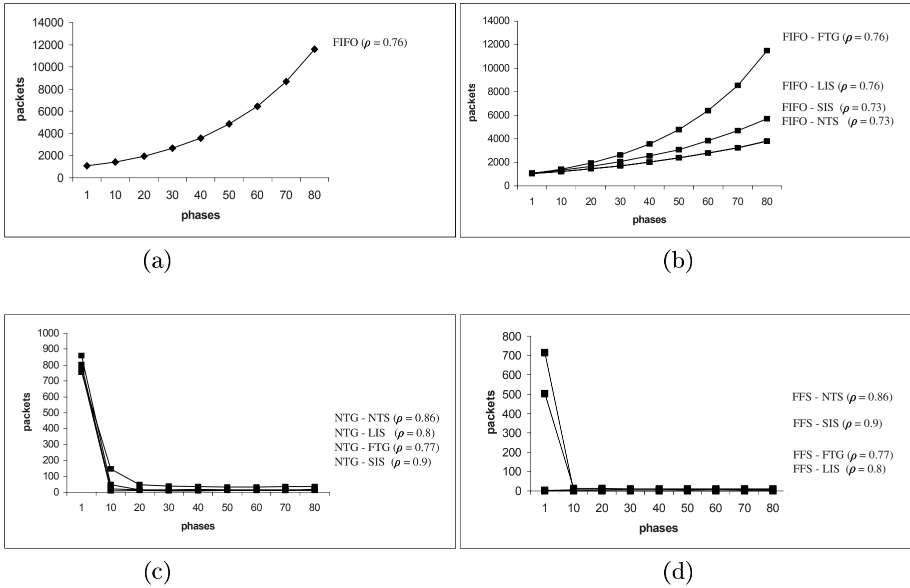


Fig. 4. Instability curves of $\mathcal{G}_3$ under a protocol or a composition of protocols: (a) FIFO, (b) FIFO-NTS, FIFO-SIS, FIFO-FTG, FIFO-LIS, (c) NTG-NTS, NTG-SIS, NTG-FTG, NTG-LIS, (d) FFS-NTS, FFS-SIS, FFS-FTG, FFS-LIS

and LIS on network $\mathcal{G}_2$ under the adversary $\mathcal{A}_2$. But, the composition of FFS protocol with NTS or SIS leads to instability.

Finally, we experiment with the effect of composing unstable protocols FIFO, NTG, FFS with universally-stable protocols NTS, FTG, LIS, SIS on network $\mathcal{G}_3$. Figure 4b depicts the total number of packets into the queues of $\mathcal{G}_3$ under the compositions of FIFO with any of NTS, FTG, LIS, SIS protocols. Furthermore, for comparison reasons, we estimate the evolution of the number of packets into the network when FIFO is used for contention-resolution on all queues of $\mathcal{G}_3$ (Figure 4a). Finally, Figures 4c, 4d depict the total number of packets into the queues of $\mathcal{G}_3$ under the compositions of NTG and FFS protocols with any of NTS, FTG, LIS, SIS protocols.

The results of the experiments on network $\mathcal{G}_3$ (Table 6), shows that the composition FIFO - NTS and FIFO - SIS is unstable with injection rate $\rho \geq 0.73$. In theory, FIFO protocol on $\mathcal{G}_3$ is unstable with $\rho \geq 0.749$ [8, Theorem 5.1]. This gives an indication that maybe FIFO instability becomes worse when we compose it with a universally stable protocol.

References

1. Alvarez, C., Blesa, M., Serna, M.: A Characterization of Universal Stability in the Adversarial Queuing model. SIAM Journal on Computing 34, 41–66 (2004)
2. Andrews, M., Awerbuch, B., Fernández, A., Kleinberg, J., Leighton, T., Liu, Z.: Universal Stability Results and Performance Bounds for Greedy Contention-Resolution Protocols. Journal of the ACM 48, 39–69 (2001)
3. Bhattacharjee, R., Goel, A.: Instability of FIFO at Arbitrarily Low Rates in the Adversarial Queueing Model. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, Cambridge, MA, USA, pp. 160–167 (2003)
4. Borodin, A., Kleinberg, J., Raghavan, P., Sudan, M., Williamson, D.: Adversarial Queueing Theory. Journal of the ACM 48, 13–38 (2001)
5. Chroni, M.: Algorithmic Topics on Stability of Heterogeneous Networks, M.Sc. Thesis, Department of Computer Science, University of Ioannina (2007)
6. Diaz, J., Koukopoulos, D., Nikoletseas, S., Serna, M., Spirakis, P., Thilikos, D.: Stability and Non-Stability of the FIFO Protocol. In: Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 48–52 (2001)
7. Herlihy, M.P., Wing, J.: Linearizability: A Correctness Condition for Concurrent Objects. In: Proceedings of the ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
8. Koukopoulos, D., Mavronicolas, M., Nikoletseas, S., Spirakis, P.: On the Stability of Compositions of Universally Stable, Greedy, Contention-Resolution Protocols. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 88–102. Springer, Heidelberg (2002)
9. Koukopoulos, D., Mavronicolas, M., Nikoletseas, S., Spirakis, P.: The Impact of Network Structure on the Stability of Greedy Protocols. Theory of Computing Systems 38, 425–460 (2005)
10. Koukopoulos, D., Nikoletseas, S., Spirakis, P.: Stability Issues in Heterogeneous and FIFO Networks under the Adversarial Queueing Model. In: Monien, B., Prasanna, V.K., Vajapeyam, S. (eds.) HiPC 2001. LNCS, vol. 2228, pp. 3–14. Springer, Heidelberg (2001)

11. Lotker, Z., Patt-Shamir, B., Rosén, A.: New Stability Results for Adversarial Queuing. SIAM Journal on Computing 33, 286–303 (2004)
12. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
13. Tsaparas, P.: Stability in Adversarial Queueing Theory, M.Sc. Thesis, Computer Science Department, University of Toronto (1997)

Simple Compression Code Supporting Random Access and Fast String Matching

Kimmo Fredriksson* and Fedor Nikitin

Department of Computer Science and Statistics, University of Joensuu
PO Box 111, FIN–80101 Joensuu, Finland
{kfredrik,fnikitin}@cs.joensuu.fi

Abstract. Given a sequence S of n symbols over some alphabet Σ, we develop a new compression method that is (*i*) very simple to implement; (*ii*) provides $O(1)$ time random access to any symbol of the original sequence; (*iii*) allows efficient pattern matching over the compressed sequence. Our simplest solution uses at most $2h + o(h)$ bits of space, where $h = n(H_0(S) + 1)$, and $H_0(S)$ is the zeroth-order empirical entropy of S. We discuss a number of improvements and trade-offs over the basic method. The new method is applied to text compression. We also propose average case optimal string matching algorithms.

1 Introduction

The aim of *compression* is to represent the given data (a sequence) using as little space as possible. This is achieved by discovering and utilizing the redundancies of the input. Some well-known compression algorithms include Huffman coding [14], arithmetic coding [26], Ziv-Lempel [28] and Burrows-Wheeler compression [6]. Recently algorithms that can compress the input sequence close to the information theoretic minimum size and still allow retrieving any symbol (actually a short substring) of the original sequence in constant time have been proposed [10,24]. These are relatively complex and not well suited for the applications we are considering. The task of *compressed pattern matching* [1,17,18] is to report all the occurrences of a given pattern in a compressed text. In this paper we give a new compression method for sequences. The main traits of the method are its extreme simplicity, good compression ratio on natural language, it provides constant time random access to any symbol of the original sequence and allows average optimal time pattern matching over the compressed sequence without decompression. We give several compression methods, having different space/time trade-offs. We analyze the compression ratios, and give several string matching algorithms to search a pattern over the compressed text. Our compression method is somewhat related to ETDC compression [3] for natural language texts.

* Supported by the Academy of Finland, grant 207022.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 203–216, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Preliminaries

Let $S[0 \ldots n-1] = s_0, s_1, s_2, \ldots, s_{n-1}$ be a sequence of symbols over an alphabet Σ of size $\sigma = |\Sigma|$. For a binary sequence $B[0 \ldots n-1]$ the function $\mathtt{rank}_b(B, i)$ returns the number of times the bit b occurs in $B[0 \ldots i]$. Function $\mathtt{select}_b(B, i)$ is the inverse, i.e. it gives the index of the ith bit that has value b. Note that for binary sequences $\mathtt{rank}_0(B, i) = i + 1 - \mathtt{rank}_1(B, i)$. Both `rank` and `select` can be computed in $O(1)$ time with only $o(n)$ bits of space in addition to the original sequence taking n bits [15,19]. It is also possible to achieve $nH_0(B) + o(n)$ total space, where $H_0(B)$ is the zero-order entropy of B [22,23], or even $H_k(B) + o(n)$ [24], while retaining the $O(1)$ query times.

The zeroth-order empirical entropy of the sequence S is defined to be

$$H_0(S) = -\sum_{s \in \Sigma} \frac{f(s)}{n} \log_2 \left(\frac{f(s)}{n} \right), \tag{1}$$

where $f(s)$ denotes the number of times s appears in S. The k-th order empirical entropy is

$$H_k(S) = -\sum_{i=0}^{n-1} p_i \log_2(p_i), \tag{2}$$

where $p_i = \text{Probability}(s_i \mid s_{i-k}, \ldots, s_{i-1})$. In other words, the symbol probabilities depend on the context they appear on, i.e. on which are the previous k symbols in S. Obviously, $H_k(S) \leq H_0(S)$.

3 Simple Dense Coding

Our compression scheme first computes the frequencies of each alphabet symbol appearing in S. Assume that the symbol $s_i \in \Sigma$ occurs $f(s_i)$ times. The symbols are then sorted by their frequency, so that the most frequent symbol comes first. Let this list be $s_{i_0}, s_{i_1}, \ldots, s_{i_{\sigma-1}}$, i.e. $i_0 \ldots i_{\sigma-1}$ is a permutation of $\{0, \ldots, \sigma-1\}$.

The coding scheme assigns binary codes with different lengths for the symbols as follows. We assign 0 for s_{i_0} and 1 for s_{i_1}. Then we use all binary codes of length 2. In that way the symbols $s_{i_2}, s_{i_3}, s_{i_4}, s_{i_5}$ get the codes $00, 01, 10, 11$, correspondingly. When all the codes with length 2 are exhausted we again increase length by 1 and assign codes of length 3 for the next symbols and so on until all symbols in the alphabet get their codes.

Theorem 1. *For the proposed coding scheme the following holds:*

1. *The binary code for the symbol* $s_{i_j} \in \Sigma$ *is of length* $\lfloor \log_2(j+2) \rfloor$.
2. *The code for the symbol* $s_{i_j} \in \Sigma$ *is binary representation of the number* $j + 2 - 2^{\lfloor \log_2(j+2) \rfloor}$ *of* $\lfloor \log_2(j+2) \rfloor$ *bits.*

Proof. Let a_ℓ and b_ℓ be indices of the first and the last symbol in alphabet Σ, which have the binary codes of length ℓ. We have $a_1 = 0$ and $b_1 = 1$. The values a_ℓ and b_ℓ for $\ell > 1$ can be defined by recurrent formulas

$$a_\ell = b_{\ell-1} + 1, \qquad b_\ell = a_\ell + 2^\ell - 1. \tag{3}$$

In order to get the values a_ℓ and b_ℓ as functions of ℓ, we first substitute the first formula in (3) to the second one and have

$$b_\ell = b_{\ell-1} + 2^\ell. \tag{4}$$

By applying the above formula many times we have a series

$$\begin{aligned} b_\ell &= b_{\ell-2} + 2^{\ell-1} + 2^\ell, \\ b_\ell &= b_{\ell-3} + 2^{\ell-2} + 2^{\ell-1} + 2^\ell, \\ &\dots \\ b_\ell &= b_1 + 2^2 + 2^3 + \dots + 2^\ell. \end{aligned}$$

Finally, b_ℓ as a function of ℓ becomes

$$b_\ell = 1 + \sum_{k=2}^{\ell} 2^k = \sum_{k=0}^{\ell} 2^k - 2 = 2^{\ell+1} - 3. \tag{5}$$

Using (3) we get

$$a_\ell = 2^\ell - 3 + 1 = 2^\ell - 2. \tag{6}$$

If j is given the length of the code for s_{i_j} is defined equal to ℓ, satisfying

$$a_\ell \leq j \leq b_\ell. \tag{7}$$

According to above explicit formulas for a_ℓ and b_ℓ we have

$$2^\ell - 2 \leq j \leq 2^{\ell+1} - 3 \quad \Longleftrightarrow \quad 2^\ell \leq j + 2 \leq 2^{\ell+1} - 1, \tag{8}$$

and finally

$$\ell \leq \log_2(j+2) \leq \log_2(2^{\ell+1} - 1), \tag{9}$$

whose solution is easily seen to be $\ell = \lfloor \log_2(j+2) \rfloor$. For the setting the second statement it is sufficient to observe that the code for the symbol $s_j \in \Sigma$ is $j - a_\ell$. By applying simple transformations we have

$$j - a_\ell = j - (2^\ell - 2) = j + 2 - 2^\ell = j + 2 - 2^{\lfloor \log_2(j+2) \rfloor}.$$

So, the second statement is also proved. □

The whole sequence is then compressed just by concatenating the codewords for each of the symbols of the original sequence. We denote the compressed binary sequence as $S' = S'[0 \dots h-1]$, where h is the number of bits in the sequence. Table 1 illustrates.

Table 1. Example of compressing the string `banana`

$S =$ `banana`	$f(\texttt{a}) = 3$	$C[\texttt{a}] = 0 = 0_2$	$T[0][0] = \texttt{a}$
$S' =$ `0001010`	$f(\texttt{n}) = 2$	$C[\texttt{n}] = 1 = 1_2$	$T[0][1] = \texttt{n}$
$D =$ `1011111`	$f(\texttt{b}) = 1$	$C[\texttt{b}] = 0 = 00_2$	$T[1][0] = \texttt{b}$

3.1 Constant Time Random Access to the Compressed Sequence

The seemingly fatal problem of the above approach is that the codes are not *prefix codes*, and we have not used any delimiting method to mark the codeword boundaries, and hence the original sequence would be impossible to obtain. However, we also create an auxiliary binary sequence $D[0 \ldots h-1]$, where h is the length of S' in bits. $D[i] = 1$ iff $S'[i]$ starts a new codeword, and 0 otherwise, see Table 1. We also need a symbol table T, such that for each different codeword length we have table of the possible codewords of the corresponding length. In other words, we have a table $T[0 \ldots \lfloor \log_2(\sigma+1) \rfloor - 1]$, such that table $T[i][0 \ldots 2^{i+1}-1]$ lists the codewords of length i. Then, given a bit-string r, $T[|r|-1][r]$ gives the decoded symbol for codeword r. This information is enough for decoding. However, D also gives us *random access to any codeword of* S'. That is, the ith codeword of S' starts at the bit position $\texttt{select}_1(D,i)$, and ends at the position $\texttt{select}_1(D,i+1)-1$. This in turn allows to access any symbol of the original sequence S in constant time. The bit-string

$$r = S'[\texttt{select}_1(D,i) \ldots \texttt{select}_1(D,i+1)-1] \tag{10}$$

gives us the codeword for the ith symbol, and hence $S[i] = T[|r|-1][r]$, where $|r|$ is the length of the bitstring r. Note that $|r| = O(\log(n))$ and hence in the RAM model of computation r can be extracted in $O(1)$ time. We call the method Simple Dense Coding (SDC). We note that similar idea (in somewhat different context) as our D vector was used in [11] with Huffman coding. However, the possibility was already mentioned in [15].

3.2 Space Complexity

The number of bits required by S' is

$$h = \sum_{j=0}^{\sigma-1} f(s_{i_j}) \lfloor \log_2(j+2) \rfloor, \tag{11}$$

and hence the average number of bits per symbol is h/n.

Theorem 2. *The number of bits required by S' is at most $n(H_0(S)+1)$.*

Proof. The zero-order empirical entropy of S is

$$-\sum_{j=0}^{\sigma-1} \frac{f(s_{i_j})}{n} \log_2\left(\frac{f(s_{i_j})}{n}\right), \tag{12}$$

and thus

$$n(H_0(S)+1) = n\sum_{j=0}^{\sigma-1}\frac{f(s_{i_j})}{n}\log_2\left(\frac{n}{f(s_{i_j})}\right)+n = \sum_{j=0}^{\sigma-1} f(s_{i_j})\left(\log_2\frac{n}{f(s_{i_j})}+1\right). \tag{13}$$

We will show that the inequality

$$\lfloor \log_2(j+2)\rfloor \le \log_2(j+2) \le \left(\log_2\left(\frac{n}{f(s_{i_j})}\right)+1\right) = \log_2\left(\frac{2n}{f(s_{i_j})}\right) \tag{14}$$

holds for every j, which is the same as

$$j+2 \le \frac{2n}{f(s_{i_j})} \iff (j+2)f(s_{i_j}) \le 2n. \tag{15}$$

Note that for $j=0$ the maximum value for $f(s_{i_j})$ is $n-\sigma+1$, and hence the inequality holds for $j=0$, $\sigma \ge 2$. In general, we have that $f(s_{i_{j+1}}) \le f(s_{i_j})$, so the maximum value for $f(s_{i_1})$ is $n/2$, since otherwise it would be larger than $f(s_{i_0})$, a contradiction. In general $f(s_{i_j}) \le n/(j+1)$, and the inequality becomes

$$(j+2)f(s_{i_j}) \le 2n \iff (j+2)n/(j+1) \le 2n \iff (j+2)/(j+1) \le 2, \tag{16}$$

which holds always. □

In general, our coding cannot achieve $H_0(S)$ bits per symbol, since we cannot represent fractional bits (as in arithmetic coding). However, if the distribution of the source symbols is not very skewed, it is possible that $h/n < H_0(S)$. This does not violate the information theoretic lower bound, since in addition to S' we need also the bit sequence D, taking another h bits. Therefore the total space we need is $2h$ bits, which is at most $2n(H_0(S)+1)$ bits. However, this can be improved.

Note that we do not actually need D, but only a data structure that can answer $\texttt{select}_1(D,i)$ queries in $O(1)$ time. This is possible using just $h' = hH_0(D)+o(n)+O(\log\log(h))$ bits of space [23]. Therefore the total space we need is only $h+h'$ bits. $H_0(D)$ is easy to compute as we know that D has exactly n bits set to 1, and $h-n$ bits to 0. Hence

$$H_0(D) \;=\; -\frac{n}{h}\log_2\left(\frac{n}{h}\right) \;-\; \frac{h-n}{h}\log_2\left(\frac{h-n}{h}\right). \tag{17}$$

This means that $H_0(D)$ is maximized when $\frac{n}{h}=\frac{1}{2}$, but on the other hand h' depends also on h. Thus, h'/h shrinks as h grows, and hence for increasing $H_0(S)$ (or for non-compressible sequences, in the worst case $H_0(S)=\log_2(\sigma)$) the contribution of $hH_0(D)$ to the total size becomes more and more negligible.

Finally, the space for the symbol table T is $\sigma\lceil\log_2(\sigma)\rceil$ bits, totally negligible in most applications. However, see Sec. 3.5 and Sec. 3.6 for examples of large alphabets.

3.3 Trade-Offs Between h and h'

So far we have used the minimum possible number of bits for the codewords. Consider now that we round each of the codeword lengths up to the next integers divisible by some constant u, i.e. the lengths are of the form $i \times u$, for $i = \{1, 2, \ldots, \lceil \log_2(\sigma) \rceil / u\}$. So far we have used $u = 1$. Using $u > 1$ obviously only increases the length of S', the compressed sequence. But the benefit is that each of the codewords in S' can start only at positions of the form $j \times u$, for $j = \{0, 1, 2, \ldots\}$. This has two consequences:

1. the bit sequence D need to store only every uth bit;
2. every removed bit is a 0 bit.

The item (2) means that the probability of 1-bit occurring increases to $\frac{n}{h/u}$. The extreme case of $u = \log_2(\sigma)$ turns D into a vector of n 1-bits, effectively making it (and S') useless. However, if we do not compress D, then the parameter u allows easy optimization of the total space required. Notice that when using $u > 1$, the codeword length becomes

$$\lfloor \log_2((2^u - 1)j + 2^u) \rfloor_u \leq \log_2((2^u - 1)j + 2^u) \tag{18}$$

bits, where $\lfloor x \rfloor_u = \lfloor x/u \rfloor u$. Then we have the following:

Theorem 3. *The number of bits required by S' is at most $n(H_0(S) + u)$.*

Proof. The theorem is easily proved by following the steps of the proof of Theorem 2. □

The space required by D is then at most $n(H_0(S) + u)/u$ bits. Summing up, the total space is optimized for $u = \sqrt{H_0(S)}$, which leads to total space of

$$n\left(H_0(S) + 2\sqrt{H_0(S)} + 1\right) + o\left(n\left(\sqrt{H_0(S)} + 1\right)\right) \tag{19}$$

bits, where the last term is for the $\texttt{select}_1$ data structure [19].

Note that $u = 7$ would correspond to byte based End Tagged Dense Code (ETDC) [3] if we do not compress D. By compressing D our space is smaller and we also achieve random access to any codeword, see Sec. 3.5.

3.4 Context Based Modelling

We note that we could easily use context based modelling to obtain h of the form $nH_k(S)$. The only problem is that for large alphabets k must be quite small, since the symbol table size is multiplied by σ^k. This can be controlled by using a k that depends on S. For example, using $k = \frac{1}{2}\log_\sigma(n)$ the space complexity is multiplied by $\sqrt{n}$, negligible for constant size alphabets.

3.5 Word Alphabets

Our method can be used to obtain very good compression ratios for natural language texts by using the σ distinct words of the text as the alphabet. By Heaps' Law [12], $\sigma = n^{\alpha}$, where n is the total number of words in the text, and $\alpha < 1$ is language dependent constant, for English $\alpha = 0.4 \ldots 0.6$. These words form a dictionary $W[0 \ldots \sigma - 1]$ of σ strings, sorted by their frequency. The compression algorithm then codes the jth most frequent word as an integer j using $\lfloor \log_2(j + 2) \rfloor$ bits. Again, the bit-vector D provides random access to any word of the original text.

As already mentioned, using $u = 7$ corresponds to the ETDC method [3]. ETDC uses 7 bits in each 8 bit byte to encode the codewords similarly as in our method. The last bit is saved for a flag that indicates whether the current byte is the last byte of the codeword. Our benefit is that as we store these flag bits into a separate vector D, we can compress D as well, and simultaneously obtain random access to the original text words.

3.6 Self-delimiting Integers

Assume that S is a sequence of integers in range $\{0, \ldots, \sigma - 1\}$. Note that our compression scheme can be directly applied to represent S succinctly, even without assigning the codewords based on the frequencies of the integers. In fact, we can just directly encode the number $S[i]$ with $\lfloor \log_2(S[i] + 2) \rfloor$ bits, and again using the auxiliary sequence D to mark the starting positions of the codewords. This approach does not need any symbol tables, so the space requirement does not depend on σ. Still, if σ and the entropy of the sequence is small, we can resort to codewords based on the symbol frequencies.

This method can be used to replace e.g. Elias δ-coding [8], which achieves

$$\lfloor \log_2(x) \rfloor + 2 \lfloor \log_2(1 + \lfloor \log_2(x) \rfloor) \rfloor + 1 \tag{20}$$

bits to code an integer x. Elias codes are self-delimiting prefix codes, so the sequence can be uniquely decompressed. However, Elias codes do not provide constant time random access to the ith integer of the sequence.

Again, we can use u to tune the space complexity of our method.

4 Random Access Fibonacci Coding

In this section we present another coding method that does not need the auxiliary sequence D. The method is a slight modification of the technique used in [11]. We also give analysis of the compression ratio achieved with this coding. Fibonacci coding uses the well-known Fibonacci numbers. Fibonacci numbers $\{f_n\}_{n=1}^{\infty}$ are the positive integers defined recursively as $f_n = f_{n-1} + f_{n-2}$, where $f_1 = f_2 = 1$. The Fibonacci numbers also have a closed-form solution called Binet's formula:

$$F(n) = (\phi^n - (1 - \phi)^n)/\sqrt{5} \tag{21}$$

where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$ is the golden ratio.

Zeckendorf's theorem [4] states that for any positive integer x there exists unique representation as

$$x = \sum_{i=0}^{k} f_{c_i} \tag{22}$$

where $c_i \geq c_{i-1} + 2$ for any $i \geq 1$. The last condition means that the sequence $\{f_{c_i}\}$ does not contain two consecutive Fibonacci numbers. Moreover, the Zeckendorf's representation of integer can be found by a greedy heuristic. The Fibonacci coding of positive integers uses the Zeckendorf's representation of integer. The code for x is a bit stream of length $\ell(x) + 1$, where

$$\ell(x) = \max\{i \mid f_i \leq x\} \tag{23}$$

The last bit in the position $\ell(x) + 1$ is set to 1. The value of ith bit is set to 1 if the Fibonacci number f_i occurred in Zeckendorf's representation and is set to 0, otherwise. By definition, the bit in position $\ell(x)$ is always set to 1. Hence, at the end of the codeword we have two consecutive ones. On the other hand two consecutive ones can not appear anywhere else within codeword. This allows us to distinguish the codewors for the separate symbols in the encoded sequence, moreover the code satisfies the prefix property.

The Fibonacci dual theorem [5] states that in Zeckendorf's representation the first Fibonacci number never occurs in the representation. In other words, we can skip the first bit reserved for the first Fibonacci number and therefore we can make the codewords shorter. Thus we can obtain the following Lemma.

Lemma 1. *The length $|c(x)|$ of the Fibonacci codeword $c(x)$ for a positive integer x satisfies $|c(x)| \leq \log_\phi(\sqrt{5}x)$.*

Proof. Obviously, the worst case for the code length is when the value x is a Fibonacci number itself. Then the code is sequence of $\ell(x) - 2$ zeroes ending with two ones. Thus, we have to estimate the value $\ell(x)$, supposing that x is Fibonacci number. Using the Binet's formula we have

$$x = (\phi^{\ell(x)} - (1 - \phi)^{\ell(x)})/\sqrt{5} \tag{24}$$

Taking logarithms from the both sides we get, after some algebra:

$$\log_2(\sqrt{5}x) = \ell(x)\log_2(\phi) + \log_2(1 + ((\phi - 1)/\phi)^{\ell(x)}). \tag{25}$$

Due to that $0 < (\phi - 1)/\phi < 1$ we have that $\log_2(\sqrt{5}x) \geq \ell(x)\log_2(\phi)$, and hence as $\phi > 1$ we have $\ell(x) \leq \log_2(\sqrt{5}x)/\log_2(\phi)$. □

Theorem 4. *The sequence S encoded with Fibonacci coding can be represented in*

$$h = n\,\frac{H_0(S) + \log_2(\sqrt{5})}{\log_2(\phi)}$$

bits.

Proof. Follows from Lemma 1. □

4.1 Random Access to Fibonacci Coded Sequences

As it was mentioned we have one attractive property of the Fibonacci code. The two consecutive ones can only appear at the end of the codeword and nowhere else (however, it is possible that the encoded *sequence* has more than two consecutive one bits, namely when codeword `11` is repeated).

If we want to start encoding from the ith symbol we should find the $(i-1)$th pair of two ones, assuming that the pairs are not overlapped. When the position j of this pair is defined we can start decoding from the position $j+2$. Thus, for our task it is enough to be able to determine the position of $(i-1)$th pair of non-overlapping ones in constant time. The query which does it we denote as $\texttt{select}_{11}$. Notice that as we do not allow the pairs to be overlapped, this query does not answer the question where the certain occurrence of substring `11` starts in the bitstream and it differs from the extended `select` query presented in [16]. The data structure for the $\texttt{select}_{11}$ query can be constructed using the same idea as for classical $\texttt{select}_1$ query solution presented by Clark [7].

The important remark which makes the Clark's approach applicable is that during construction of the block directories every sequence `11` of interest is included entirely in range. It allows us to use look-up tables, as the situation of the sequence `11` belonging into two ranges at the same time is impossible. We omit the details, but by following the Clark's construction, we can build a data structure taking $o(h)$ bits of space (in additional to the original sequence) and supporting $\texttt{select}_{11}$ queries in $O(1)$ time.

5 Fast String Matching

We now present several efficient string matching algorithms working on the compressed texts. We assume SDC here, altough the algorithms work with minor modification in Fibonacci coded texts as well. The basic idea of all the algorithms is that we compress the pattern using the same method and dictionary as for compressing the text, so as to be able to directly comprare a text substring against the pattern. We denote the compressed text and the auxiliary bitvector as S' and $D_{S'}$, both consisting of h bits. For clarity of presentation, we assume that $u = 1$. Likewise, the compressed sequences for the pattern are denoted as P' and $D_{P'}$, of m bits.

5.1 BMH Approach

The well-known Boyer-Moore-Horspool algorithm (BMH) [13] works as follows. The pattern P is aligned against a text window $S[i-m+1 \ldots i]$ The invariant is that every occurrence of P ending before the position i is already reported. Then the symbol $P[m-1]$ (i.e. the last symbol of P) is compared against $S[i]$. If they match, the whole pattern is compared against the window, and a possible match

Alg. 1 SearchBMH(T', D_T, n, P', D_P, m)

```
1   b ← O(log(m))
2   for i ← 0 to (1 << b) − 1 do shift[i] ← m
3   for i ← 1 to b − 1 do
4       c ← P[0 ... i − 1]
5       for j ← 0 to (1 << (b − i)) − 1 do shift[(c << (b − i)) | j] ← m − i
6   for i ← 0 to m − b − 1 do shift[P'[i ... i + b − 1]] = m − i − b
7   a ← P'[m − b ... m − 1]
8   occ ← 0
9   for i ← m − 1 to n − 1 do
10      c ← T'[i − b + 1 ... i]
11      if a = c AND D_T[i − m + 1 ... i] = D_P AND T'[i − m + 1 ... i] = P' then occ ← occ + 1
12      i ← i + shift[c]
13  return occ
```

is reported. Then the next window to be compared is $S[i-m+1+\mathit{shift} \ldots i+\mathit{shift}]$ (regardless of whether $S[i]$ was equal to $P[m-1]$ or not), where *shift* is computed as

$$\mathit{shift} = m - \max(j \mid P[j] = S[i], 0 \leq j < m-1) \tag{26}$$

If $S[i]$ does not occur in P, then the shift value is m. The shift function is easy to compute at the preprocessing time, needing $O(\sigma + m)$ time and $O(\sigma)$ space. The algorithm is very simple to implement and one of the most efficient algorithms in practice for reasonably large alphabets (say, $\sigma > m$), when the average case time approaches the best case time, i.e. $O(n/m)$. In our case, however, we have binary alphabet and the shift values yielded are close to 1. However, we can form a "super-alphabet" from the consecutive bits. That is, we can read b bits at a time and treat the bitstring as a symbol from an alphabet of size 2^b. I.e. we read the bitstring $S'[i-b+1\ldots i]$ and compute the shift function so as to align this bitstring against its right-most occurrence in P'. If such occurrence is not found, we compare the suffixes of $S'[i-b+1\ldots i]$ against the prefixes of $P'[0\ldots b]$. If no occurrence is still found, the shift is again m (bits). We must still verify any occurrence by comparing $D_{S'}[i-m+1\ldots i]$ against $D_{P'}$ to check that the codewords are synchronized. Alg. 1 shows complete pseudo code.

Theorem 5. *Alg. 1 runs in $O(m^2 + h/m)$ average time for the optimal b.*

Proof. The average time of Alg. 1 clearly depends on the parameter b. If $S'[i-b+1\ldots i]$ does not occur in P', then the shift is at least $m-b+1$ bits. Note that in RAM model of computation obtaining the bitstring and thus computing the shift takes $O(1)$ time as long as $b = O(\log(n))$. The total time needed for these cases (1) is thus $O(h/(m-b))$. Now, assume that if $S'[i-b+1\ldots i]$ occurs in P' we verify the whole pattern and shift only by one bit. The total time needed for these cases (2) is at most $O(hm)$ (actually only $O(hm/w)$ time, as we can compare w bits at the time, where w is the number of bits in a machine word), but only $O(h)$ on average. We therefore want to choose b so that the probability p of case (2) is low enough. The total time is therefore $O(h/(m-b) + phm)$. Assuming that the bit values have uniform distribution, $p = 1/2^b$, and the total time is optimized for

$$h/(m-b) > hm/2^b \quad \Rightarrow \quad b = \Omega(\log(m^2)), \tag{27}$$

and the average case time then becomes $O(h/m)$. The preprocessing time is $O(2^b + m)$ which is $O(m^2)$ for $b = \log_2(m^2)$. □

We note that this breaks the lower bound of $O(h \log(m)/m)$, which is based on comparison model [27], and is thus optimal. However, our method is not based on comparing single symbols and we effectively avoid the $\log(m)$ term by "comparing" b symbols at a time. On the other hand, it is easy to see that increasing b beyond $O(\log(m))$ does not improve our algorithm. Finally, note that other BMH variants, such as the one by Sunday [25], could be generalized just as easily.

5.2 Shift-Or and BNDM

The two well-known bit-parallel string matching algorithms Shift-Or [2] and BNDM [20] can be directly applied to our case, and even simplified: as already noted in [20], the preprocessing phase can be completely removed, as for binary alphabets the pattern itself and its bit-wise complement can serve as the preprocessed auxiliary table the algorithms need. However, we still need to verify the occurrences using the $D_{P'}$ sequence. The average case running times of Shift-Or and BNDM become $O(h)$ and $O(h \log(m)/m)$ for $m \leq w$. For longer patterns these must be multiplied by $\lceil m/w \rceil$. We omit the details for lack of space. However, we note that the "superalphabet" trick of the previous section works for these two algorithms as well (see also [9]). For example, we can improve BNDM by precomputing the steps taken by the algorithm by the first b bits read in a text window, and at the search phase we use a look-up table to perform the steps in $O(1)$ time, and then continue the algorithm normally. The average case time of BNDM is improved to $O(h \lceil \log(m)/b \rceil / m)$, and we see that $b = \log(m)$ gives again $O(h/m)$ average time. Preprocessing time and space become $O(m)$.

6 Experimental Results

We have run experiments to evaluate the performance of our algorithms. The experiments were run on Celeron 1.5GHz with 512Mb of RAM, running GNU/Linux operating system. We have implemented all the algorithms in C, and compiled with `gcc 4.1.1`.

The test files are summarized in Table 2 (a), the files are from Silesia corpus[1] and Canterbury corpus[2]. We used a word based model [18]: we have two dictionaries, one for the text words and the other for "separators", where separator is defined to be any substring between two words. As there is strictly alternating order between the two, decompressing is easy as far as we know whether the text starts with a word or a separator. We used `zlib` library[3] to compress the dictionaries. We also experimented with a so called "space-less model", but omit the results for a lack of space.

Table 2 (b) gives the compression ratios for several different methods. The Huffman compression algorithm uses two dictionaries, while ETDC uses the

---

[1] `http://www-zo.iinf.polsl.gliwice.pl/~sdeor/corpus.htm`
[2] `http://corpus.canterbury.ac.nz/`
[3] `www.zlib.org`

Table 2. Test files and compression ratios

(a)	Test files				
Name	Type	Size	σ (words+separators)	words	H_0(words)
dickens	English text	10,192,446 B	34,381 + 1,071	1,819,394	9.92 bits
world192.txt	English text	2,473,400 B	22,917 + 498	343,139	10.91 bits
samba	source code	6,760,204 B	29,822 + 15,544	924,640	10.40 bits
xml	xml source	5,303,867 B	19,582 + 1,495	847,806	9.10 bits

(b)	Compression ratios								
File	gzip -9	bzip2 -9	SDC	SDC W	FibC	FibC W	Huffman	ETDC	H_0
dickens	37.7%	27.4%	35.3%	29.3%	31.5%	25.4%	28.3%	32.9%	26.2%
world192.txt	29.1%	19.7%	35.5%	29.3%	31.6%	25.3%	29.0%	34.4%	24.4%
samba	20.1%	16.3%	36.1%	24.9%	32.2%	21.4%	30.3%	38.4%	27.4%
xml	12.3%	8.0%	33.0%	24.5%	30.6%	21.6%	28.7%	38.6%	26.4%

(c)	Compression ratios with and w/o `select` data structure							
	with `select`		with `select` only for words		w/o `select`, word stream only SDC			
File	SDC	FibC	SDC	FibC	$u = 1$	$u = 2$	$u = 3$	$u = 4$
dickens	39.1%	35.4%	37.1%	33.5%	29.3%	25.8%	25.2%	25.4%
world192.txt	38.6%	35.2%	37.0%	33.7%	29.3%	25.7%	24.9%	25.1%
samba	39.1%	35.4%	37.6%	33.8%	24.9%	21.7%	21.1%	21.3%
xml	36.6%	34.4%	34.7%	32.6%	24.5%	21.9%	21.6%	21.9%

space-less model. Note that SDC with $u = 7$ and spaceless model would achieve the same ratio. H_0 denotes the empirical entropy using the model of two separate dictionaries. SDC W and FibC W columns give the ratios using only the word stream (i.e. excluding the separators stream). This gives the text size for our search algorithms, since we ran them only for the words.

Table 2 (c) shows the compression ratios for SDC and FibC including the size of the `select` data structures. The values are for both streams (words and seperators), and for word stream only. We used the `darray` method [21]. For SDC coding this can be directly applied on D vector. For FibC this needs some modifications, but these are quite easy and straight-forward. The table shows also the effect of the parameter u for SDC. We show only the effect on word stream. In general we can, and should, optimize the parameter individually for words and separators, since the entropy for separators is usually much smaller. The optimum value is $u = 3$ in all cases, as can be deduced from Table 2 (a), i.e. the optimum is $\sqrt{H_0(\text{words})}$.

Fig. 1 shows the search performance using *dickens* file. We compared our algorithms against the BMH algorithm on the original uncompressed text. We used patterns consisting of $1 \ldots 4$ words and 300 patterns of each length randomly picked from the text. The compressed pattern lengths were about 6, 12, 18 and 25 bits, correspondingly. Shift-Or (SO) is quite slow, as expected (as the shift is always just 1 bit). However, BNDM, BNDMB (same as BNDM but using the parameter b) and FBMH (our BMH variant running on compressed texts) achieve reasonably good performance, altough they lose to plain BMH, which has very simple implementation. For FBMH we used $b = m$ for $m \leq 10$ and $b = 10$ for larger m. For BNDMB we used $b = 2\log_2(m)$. We feel that the performance of searching in compressed texts could still be improved. In particular, using

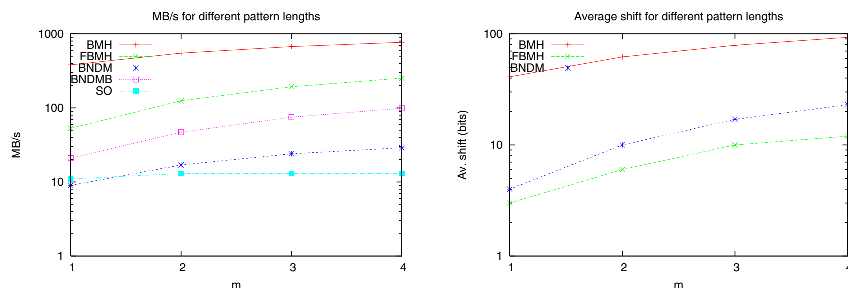


Fig. 1. Search performance. Left: MB/second processed by different algorithm. Right: the average shift in bits. The x-axis (m) is the pattern length in *words*.

$u = 8$ allows us to use plain byte based BMH algorithm, with the exception that we have to verify the occurrences with the D vector.

7 Conclusions

We have presented a simple compression schemes that allow constant time access to any symbol of the original sequence. The method gives good compression ratio for natural language texts, and allows average-optimal time string matching without decompression.

References

1. Amir, A., Benson, G.: Two-dimensional periodicity and its applications. In: Proceedings of SODA'92, pp. 440–452 (1992)
2. Baeza-Yates, R.A., Gonnet, G.H.: A new approach to text searching. Commun. ACM 35(10), 74–82 (1992)
3. Brisaboa, N., Iglesias, E., Navarro, G., Paramá, J.: An efficient compression code for text databases. In: Sebastiani, F. (ed.) ECIR 2003. LNCS, vol. 2633, pp. 468–481. Springer, Heidelberg (2003)
4. Brown, J.L.: Zeckendorf's theorem and some applications. Fib. Quart. vol. 2, pp. 163–168
5. Brown, J.L.: A new characterization of the Fibonacci numbers. Fib. Quart. 3, 1–8 (1965)
6. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
7. Clark, D.R.: Compact Pat Trees. PhD thesis, University of Waterloo, Ontario, Canada (1998)
8. Elias, P.: Universal codeword sets and representation of the integers. IEEE Transactions on Information Theory 21(2), 194–203 (1975)
9. Fredriksson, K.: Shift–or string matching with super-alphabets. Information Processing Letters 87(1), 201–204 (2003)

10. González, R., Navarro, G.: Statistical encoding of succinct data structures. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 295–306. Springer, Heidelberg (2006)
11. Grabowski, S., Navarro, G., Przywarski, R., Salinger, A., Mäkinen, V.: A simple alphabet-independent FM-index. International Journal of Foundations of Computer Science (IJFCS) 17(6), 1365–1384 (2006)
12. Heaps, H.S.: Information retrieval: theoretical and computational aspects. Academic Press, New York (1978)
13. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. Exp. 10(6), 501–506 (1980)
14. Huffman, D.A.: A method for the construction of minimum redundancy codes. Proceedings of I.R.E 40, 1098–1101 (1951)
15. Jacobson, G.: Succinct static data structures. PhD thesis, Carnegie Mellon University (1989)
16. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science, Special issue on "The Burrows-Wheeler Transform and its Applications". To appear (2006)
17. Manber, U.: A text compression scheme that allows fast searching directly in the compressed file. ACM Trans. Inform. Syst. 15(2), 124–136 (1997)
18. Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast and flexible word searching on compressed text. ACM Transactions on Information Systems (TOIS) 18(2), 113–139 (2000)
19. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
20. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM Journal of Experimental Algorithmics (JEA), vol. 5(4) (2000)
21. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proceedings of ALENEX'07, ACM Press, New York (2007)
22. Pagh, R.: Low redundancy in static dictionaries with o(1) worst case lookup time. In: Wiedermann, J., van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 595–604. Springer, Heidelberg (1999)
23. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of SODA'02, pp. 233–242. ACM Press, New York (2002)
24. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proceedings of SODA'06, pp. 1230–1239. ACM Press, New York (2006)
25. Sunday, D.M.: A very fast substring search algorithm. Commun. ACM 33(8), 132–142 (1990)
26. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Communications of the ACM 30(6), 520 (1987)
27. Yao, A.C.: The complexity of pattern matching for a random string. SIAM J. Comput. 8(3), 368–387 (1979)
28. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory 23, 337–343 (1977)

Engineering a Compressed Suffix Tree Implementation

Niko Välimäki[1], Wolfgang Gerlach[2], Kashyap Dixit[3], and Veli Mäkinen[1,⋆]

[1] Department of Computer Science, University of Helsinki, Finland
{nvalimak,vmakinen}@cs.helsinki.fi
[2] Technische Fakultät, Universität Bielefeld, Germany
wgerlach@cebitec.uni-bielefeld.de
[3] Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur, India
kdixit@iitk.ac.in

Abstract. Suffix tree is one of the most important data structures in string algorithms and biological sequence analysis. Unfortunately, when it comes to implementing those algorithms and applying them to real genomic sequences, often the main memory size becomes the bottleneck. This is easily explained by the fact that while a DNA sequence of length n from alphabet $\Sigma = \{A, C, G, T\}$ can be stored in $n \log |\Sigma| = 2n$ bits, its suffix tree occupies $O(n \log n)$ bits. In practice, the size difference easily reaches factor 50.

We report on an implementation of the compressed suffix tree very recently proposed by Sadakane (*Theory of Computing Systems*, in press). The compressed suffix tree occupies space proportional to the text size, i.e. $O(n \log |\Sigma|)$ bits, and supports all typical suffix tree operations with at most $\log n$ factor slowdown. Our experiments show that, e.g. on a 10 MB DNA sequence, the compressed suffix tree takes 10% of the space of normal suffix tree. At the same time, a representative algorithm is slowed down by factor 30.

Our implementation follows the original proposal in spirit, but some internal parts are tailored towards practical implementation. Our construction algorithm has time requirement $O(n \log n \log |\Sigma|)$ and uses closely the same space as the final structure while constructing it: on the 10 MB DNA sequence, the maximum space usage during construction is only 1.4 times the final product size.

1 Introduction

Myriad non-trivial combinatorial questions concerning strings turn out to have efficient solutions via extensive use of *suffix trees* [2]. As a theoretical tool, suffix trees have a fundamental role in plethora of algorithmic results in the area of string matching and sequence analysis. *Bioinformatics* is a field where suffix trees

⋆ Funded by the Academy of Finland under grant 108219.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 217–228, 2007.
© Springer-Verlag Berlin Heidelberg 2007

would seem to have the strongest practical potential; unlike the natural language texts formed by words and delimiters (enabling specialized data structures like inverted files), biological sequences are streams of symbols without any predefined word boundaries.. Suffix trees treat any substring equally, regardless of it being a word or not. This perfect synergy has created a vast literature describing suffix tree -based algorithms for sequence analysis problems, see e.g. [13]. Several implementations exist as well, like `STRMAT`, `WOTD`, `LIBSTREE`, and `MUMMER`[1], to name a few.

Unfortunately, the theoretically attractive properties of suffix trees do not always meet the practical realm. A bottleneck to the wide-spread use of suffix trees in Bioinformatics is their immense space consumption. Even for a reasonable size genomic sequence of 100 MB, its suffix tree may require 5 GB of main memory. This phenomenon is not just a consequence of constant factors in the implementation of the structure, but rather an asymptotic effect. When examined more carefully, one notices that a sequence of length n from an alphabet Σ requires only $n \log |\Sigma|$ *bits* of space, whereas its suffix tree requires $O(n \log n)$ bits. Hence, the space requirement is by no means linear when measured in bit-level.

The size bottleneck of suffix trees has made the research turn into looking for more space-economic variants of suffix trees. One popular alternative is the *suffix array* [20]. It basically removes the constant factor of suffix trees to 1, as what remains from suffix trees is a lexicographically ordered array of starting positions of suffixes in the text. That occupies $n \log n$ bits. Many tasks on suffix trees can be simulated by $\log n$ factor slowdown using suffix arrays. With three additional tables, suffix arrays can be enhanced to support typical suffix tree operations without any slowdown [1].

A recent twist in the development of full-text indexes is the use of *abstract data structures*; the operations supported by a data structure are identified and the best possible implementation is sought for that supports those operations. This line of development has led to *compressed suffix arrays* [12,9] (see [24] for more references). These data structures take, in essence, $n \log |\Sigma|(1 + o(1))$ bits of space, being asymptotically space-optimal. For compressible sequences they take even less space. More importantly, they simulate suffix array operations with logarithmic slowdowns. These structures are also called *self-indexes* as they do not need the text to function; the text is actually represented compressed within the index.

Very recently Sadakane [25] extended the abstract data structure concept to cover suffix trees, identifying typical operations suffix trees are assumed to possess. Some of these operations, like navigating in a tree, were already extensively studied by Munro, Raman, and Rao [22]. In addition to these navigational operations, suffix trees have several other useful operations such as suffix links, constant time lowest common ancestor (lca) queries, and pattern search capabilities. Sadakane developed a fully functional suffix tree structure by combining

[1] http://www.cs.ucdavis.edu/˜gusfield/strmat.html, http://bibiserv.techfak.uni-bielefeld.de/wotd/, http://www.cl.cam.ac.uk/˜cpk25/libstree/, http://sourceforge.net/projects/mummer/

compressed suffix arrays with several other non-trivial new structures. Each operation was supported by at most $\log n$ slowdown, often the slowdown being only constant. The space requirement was shown to be still asymptotically optimal, more accurately, $|CSA| + 6n + o(n)$ bits, where $|CSA|$ is the size of the compressed suffix array used.

This paper studies an implementation of Sadakane's compressed suffix tree. We implemented the structure following closely the original proposal [25]. In addition, we considered the issue of space-efficient construction, studying the following subtasks: (1) How to construct the Burrows-Wheeler transform on which the compressed suffix arrays are based on; (2) storing sampled text/suffix array positions; (3) direct construction of compressed longest common prefix information, and (4) construction of balanced parentheses representation of suffix tree directly from compressed suffix array. Tasks (1), (3) and (4) have been considered in [15] and later improved in [16] so as to obtain an $O(n \log^{\epsilon} n)$ time algorithm to construct compressed suffix trees, where $\epsilon > 0$. Task (2) is related to our choice of implementing compressed suffix arrays using structures evolved from FM-index [9], and is tackled in this paper. Also for task (3) our solution variates slightly from [15] as we build on top of the suffixes-insertion algorithm [6] and they build on top of the post-order traversal algorithm of [17]. The final time-requirement of our implementation is $O(n \log n \log |\Sigma|)$, being reasonably close to the best current theoretical result [16].

The outline of the article is as follows. Section 2 gives the basic definitions and a very cursory overview of Sadakane's structure. Section 3 explains how we implemented compressed suffix arrays (related to task (1)) and provides a solutions to task (2). Section 4 describes the solution hinted in [15] for task (3). Section 5 gives an overview of balanced parentheses and describes our construction algorithm, solving task (4). Section 6 explains how we implemented the lowest common ancestor structure by adding a space-time tradeoff parameter. We conclude with some illustrative experimental results in Sect. 7.

The software package can be downloaded from `http://www.cs.helsinki.fi/group/suds/cst/`. Also a technical report is available there that contains the full implementation details that are omitted here for the lack of space.

2 Preliminaries

A *string* $T = t_1 t_2 \cdots t_n$ is a sequence of *characters* from an ordered *alphabet* Σ. A *substring* of T is any string $T_{i\ldots j} = t_i t_{i+1} \cdots t_j$, where $1 \leq i \leq j \leq n$. A *suffix* of T is any substring $T_{i\ldots n}$, where $1 \leq i \leq n$. A *prefix* of T is any substring $T_{1\ldots j}$, where $1 \leq j \leq n$.

Definition 1. *(Adopted from [13]) The* keyword trie *for set $\mathcal{P}$ of strings is a rooted directed tree $\mathcal{K}$ satisfying three conditions: (1) Each edge is labeled with exactly one character; (2) any two edges out of the same node have distinct labels; (3) every pattern P of $\mathcal{P}$ maps to some node v of $\mathcal{K}$ such that the characters on*

the path from the root of $\mathcal{K}$ to v spell out P, and every leaf of $\mathcal{K}$ is mapped to by some string in $\mathcal{P}$.

Definition 2. *The* suffix trie *of text T is a keyword trie for set $\mathcal{S}$, where $\mathcal{S}$ is the set of all suffixes of T.*

Definition 3. *The* suffix tree *of text T is the* path-compressed *suffix trie of T, i.e., a tree that is obtained by representing each maximal non-branching path of the suffix trie as a single edge labeled by the catenation of the labels in the corresponding edges of the suffix trie. The* edge labels *of suffix tree correspond to substrings of T; each edge can be represented as a pair (l, r), such that $T_{l...r}$ gives the label.*

A *path label* of a node v is the catenation of edge labels from root to v. Its length is called *string depth*. The number of edges from root to v is called *node depth*. The *suffix link* $sl(v)$ of an internal node v with path label $x\alpha$, where x denotes a single character and α denotes a possibly empty substring, is the node with path label α.

A typical operation on suffix trees is the *lowest common ancestor* query, which can be used to compute the *longest common extension* $lce(i, j)$ of arbitrary two suffixes $T_{i...n}$ and $T_{j...n}$: Let v and w be the two leaves of suffix tree have path labels $T_{i...n}$ and $T_{j...n}$, respectively. Then the path label α of the lowest common ancestor node of v and w is the longest prefix shared by the two suffixes. We have $lce(i, j) = |\alpha|$.

The following abstract definition captures the above mentioned typical suffix tree operations.

Definition 4. *An* abstract suffix tree *for a text supports the following operations:*

1. *$root()$: returns the root node.*
2. *$isleaf(v)$: returns Yes if v is a leaf, and No otherwise.*
3. *$child(v, c)$: returns the node w that is a child of v and the edge (v, w) begins with character c, or returns 0 if no such child exists.*
4. *$sibling(v)$: returns the next sibling of node v.*
5. *$parent(v)$: returns the parent node of v.*
6. *$edge(v, d)$: returns the d-th character of the edge-label of an edge pointing to v.*
7. *$depth(v)$: returns the string depth of node v.*
8. *$lca(v, w)$: returns the lowest common ancestor between nodes v and w.*
9. *$sl(v)$: returns the node w that is pointed to by the suffix link from v.*

2.1 Overview of Compressed Suffix Tree

Sadakane [25] shows how to implement each operation listed in Def. 4 by means of a sequence of operations on (1) compressed suffix array, (2) *lcp*-array [2], (3) balanced parentheses representation of suffix tree hierarchy, and (4) a structure for *lca*-queries. In the following sections we explain how we implemented those structures.

[2] Sadakane [25] uses name *Height*-array.

3 Compressed Suffix Array

Suffix array is a simplified version of suffix tree; it only lists the suffixes of the text in lexicogaphic order. Let $SA[1 \ldots n]$ be a table such that $T_{SA[i] \ldots n}$ gives the i-th smallest suffix in lexicographic order. Notice that this table can be filled by a depth-first traversal on suffix tree following its edges in lexicogaphic order.

As the array SA takes $n \log n$ bits, there has been considerable effort in building *compressed suffix arrays* to reduce its space requirement, see [24]. The following captures typical suffix array operations on an abstract level.

Definition 5. *An* abstract suffix array *for a text T supports the following operations:*

- *$lookup(i)$: returns $SA[i]$,*
- *$inverse(i)$: returns $j = SA^{-1}[i]$, defined such that $SA[j] = i$,*
- *$\Psi(i)$: returns $SA^{-1}[SA[i]+1]$, and*
- *$substring(i, l)$: returns $T[SA[i] \ldots SA[i]+l-1]$.*

3.1 Our Implementation

We used *Succinct Suffix Array* (SSA) of [18] to implement the abstract suffix array operations. The base structure is the *wavelet tree* [11] build on the *Burrows-Wheeler transform* [3]. The Burrows-Wheeler transform T^{bwt} is defined as $T^{bwt}[i] = T_{SA[i]-1}$ (where $SA[i]-1 = SA[n]$ when $SA[i] = 1$). A property of T^{bwt} used in compressed suffix arrays is so-called LF-mapping: $LF(i) = i'$ such that $SA[i'] = SA[i] - 1$.

It can be shown [9] that LF-mapping can computed by the means of T^{bwt}:

Lemma 1 ([9]). *Let $c = T^{bwt}[i]$. Then*

$$LF(i) = C[c] + rank_c(T^{bwt}, i), \quad (1)$$

where $C[c]$ is the the number of positions of T^{bwt} containing a character smaller than c and $rank_c(T^{bwt}, i)$ tells how many times character c occurs upto position i in T^{bwt}.

Table $C[1 \ldots |\Sigma|]$ can be stored as is in $|\Sigma| \log n$ bits of space, and space-efficient data structures built for storing $rank_c$-function values. For example, a simplified version of the wavelet tree (see [18, Sect. 5]) stores those values in $n \log |\Sigma|(1 + o(1))$ bits so that each $rank_c$ value (as well as value $T^{bwt}[i]$) can be computed in $O(\log |\Sigma|)$ time.

Rest of the abstract suffix array operations can be supported by storing sampled suffix array values (and sampled inverse suffix array values) and using LF-mapping to simulate those operations. These additional structures occupy together $\frac{2n}{R} \log n + n(1 + o(1)$ bits, where R is the sample rate used. See the full version of this article for details or consult [24], where these are explained for $lookup()$ and $substring()$; $\Psi()$ and $inverse()$ are analogous.

In our implementation, we use the Huffman-tree shape as advised in [18], so that the structure takes overall $\frac{2n}{R}\log n + n(H_0+2)(1+o(1))$ bits of space and supports all the abstract suffix array operations in $O(R \cdot H_0)$ average time. (Worst case $O(R \cdot \log n)$.) Here H_0 is the *zeroth order entropy* of T. Recall that $H_0 \leq \log|\Sigma|$. Fixing any $R = \Omega(\frac{\log n}{\log|\Sigma|})$, the structure takes $O(n \log|\Sigma|)$ bits.

Space-efficient Construction via Dynamic Structure. The construction of the structure is done in two phases. First the Burrows-Wheeler transform is constructed, then the additional structures.

The first phase can be executed in $O(n \log n \log|\Sigma|)$ time and using $nH_0 + o(n \log|\Sigma|)$ bits of space by using the dynamic self-index explained in [19]. We implemented the simplified version that uses $O(n \log|\Sigma|)$ bits: Instead of using the more complicated solution to solve *rank*-queries on dynamic bitvectors, we used the $O(n)$ bits structure of [4] (see also [19, Sect. 3.2]). Using this inside the dynamic wavelet trees of [19], one obtains the claimed result (see the paragraph just before Sect. 6 in [19]). The result is actually a dynamic wavelet tree of the Burrows-Wheeler transform supporting $rank_c$-queries in $O(\log n \log|\Sigma|)$ time. This is easily converted into a static structure of the original SSA (in time linear in the size of the structure) that supports $rank_c$-queries in $O(\log|\Sigma|)$ time. In our implementation, we use the Huffman-shaped wavelet tree to improve the space to $O(nH_0)$ bits. This conversion is also easily done by extracting the Burrows-Wheeler transform from the dynamic wavelet tree with a depth-first traversal and creating the Huffman-balanced static wavelet tree instead as in [18].

The rest of the structures to support abstract suffix array operations can be constructed afterward in $O(n \log|\Sigma|)$ time using LF-mapping. We leave the details for the full version.

4 *lcp*-Array

Array $lcp[1 \dots n-1]$ is used to store the longest common prefix information between consecutive suffixes in the lexicographic order. That is, $lcp[i] = |prefix(T_{SA[i]\dots n}, T_{SA[i+1]\dots n})|$, where $prefix(X,Y) = x_1 \cdots x_j$ such that $x_1 = y_1$, $x_2 = y_2$, $\dots$, $x_j = y_j$, but $x_{j+1} \neq y_{j+1}$. Sadakane [25] describes a clever encoding of the *lcp*-array that uses $2n+o(n)$ bits. The encoding is based on the fact that values $i+lcp[i]$ are increasing when listed in the text position order; sequence $S = s_1, \dots, s_{n-1} = 1+lcp[SA^{-1}[1]], 2+lcp[SA^{-1}[2]], \dots, n-1+lcp[SA^{n-1}[n-1]]$ is increasing.

To encode the increasing list S, it is enough to encode each $\texttt{diff}(i) = s_i - s_{i-1}$ in unary: $0^{\texttt{diff}(i)}1$, where we assume $s_0 = 0$ and 0^d denotes repetition of 0-bit d-times. This encoding, call it H, takes at most $2n$ bits. We have the connection $\texttt{diff}(k) = select_1(H,k) - select_1(H,k-1) - 1$, where $select_1(H,k)$ gives the position of the k-th 1-bit in H. Bitvector H can be preprocessed to answer $select_1(H,k)$-queries in constant time using $o(|H|)$ bits extra space [21].

Computing $lcp[i]$ can now be done as follows. Compute $k = SA[i]$ using $lookup(i)$. Value $lcp[i]$ equals $select_1(H,k) - k$.

Kasai et al. [17] gave a linear time algorithm to construct *lcp*-array given SA. One can easily modify Kasai et al. algorithm to directly give H [15]. The construction uses no extra memory in addition to text, compressed suffix array, and the outcome of size $2n + o(n)$ bits. Using the compressed suffix array explained earlier in this paper, the time requirement is $O(n \log n)$.

5 Balanced Parentheses

The *balanced parenthesis* representation P of a tree is produced by a preorder traversal printing $'('$ whenever a node is visited the first time, and printing $')'$ whenever a node is visited the last time [22]. Letting $'(' = 1$ and $')' = 0$, the sequence P takes $2u$ bits on a tree of u nodes. A suffix tree of n leaves can have at most $n - 1$ internal nodes, and hence its balanced parenthesis representation takes at most $4n$ bits.

Munro, Raman, and Rao [22] explain how to simulate tree traversal by means of P. After building several structures of sublinear size, one can go e.g. from node to its first child, from node to its next sibling, and from node to its parent, each in constant time. Sadakane [25] lists many other operations that are required in his compressed suffix tree. All these navigational operations can be expressed as combinations of the following functions: $rank_p$, $select_p$, $findclose$, and *enclose*. Here p is a constant size bitvector pattern, e.g. 10 expresses an open-close parenthesis pair. Function $rank_p(P, i)$ returns the number of occurrences of p in P upto position i. Function $select_p(P, j)$ returns the position of the j-th occurrences of p in P. Function $findclose(P, i)$ returns the position of the matching closing parenthesis for the open parenthesis at position i. Function $enclose(P, i)$ returns the open parenthesis position of the parent of the node whose open parenthesis is at position i.

5.1 Our Implementation

We used the existing *rank* and *select* implementations that are explained and experimented in [10]. We leave for the full version the explanation how we modified these solutions to the case of short patterns p, as the original implementations assume $p = 1$. For $findclose$ and *enclose* we used Navarro's implementations explained in [23] that are based on [22].

5.2 Space-Efficient Construction Via LCP Information

To build balanced parentheses sequence of suffix tree space-efficiently one cannot proceed naively; doing preorder traversal on a pointer-based suffix tree requires $O(n \log n)$ bits of extra memory. We consider a new approach that builds the parentheses sequence incrementally. Very similar algorithm is already given in [15], and hence we only sketch the main idea and differences.

Recall from [6, Theorem 7.5, p. 97] the *suffixes-insertion* algorithm to construct suffix tree from LCP information: The algorithm adds suffixes in

lexicographic order into a tree, having the keyword tree of suffixes $T_{SA[1]...n}$, $T_{SA[2]...n}, \ldots, T_{SA[i]...n}$ ready after i-th step. Suffix $T_{SA[i+1]...n}$ is then added after finding bottom-up from the rightmost path of the tree the correct insertion point. That is, the *split node* v closest to the rightmost leaf (corresponding to suffix $T_{SA[i]...n}$) whose string depth is smaller or equal to $lcp[i]$ is sought for. If the depth is equal, then a new leaf (corresponding to suffix $T_{SA[i+1]...n}$) is created as its child. Otherwise, its outgoing rightmost edge is split, a new internal node is inserted in between, and the leaf corresponding to suffix $T_{SA[i+1]...n}$ is added as its rightmost child.

To obtain a space-efficient version of the algorithm, we maintain the balanced parentheses representation of the tree at each step. Unfortunately, the parentheses structure does not change sequentially, so we need to maintain it using a dynamic bitvector allowing insertions of bits (open/close parentheses) inside it. Such bitvector can be maintained using $O(n)$ bits of space so that accessing the bits and inserting/deleting takes $O(\log n)$ time [4,19]. In addition to the balanced parentheses to store the tree hierarchy, we need more operations on the rightmost path; we need to be able to virtually browse the rightmost path from leaf to root as well as to compute the the string depth of each node visited. It happens that the string and node depths are monotonic on the rightmost path, and during the algorithm one only needs to modify them from the tail. Such monotonic sequences of numbers, whose sum is $O(n)$, can be stored in $O(n)$ bits using integer codes like Elias δ-code [7]. We leave the details to the full version. Hence we can construct the balanced parentheses sequence in $O(n \log n)$ time using $O(n)$ bits working space.

The difference to Hon and Sadakane algorithm [15] is mainly on the conceptual level. They build on top of an algorithm in [17] that simulates the post-order traversal of suffix tree given the lcp-values. When making that algorithm space-efficient, the end result is very close to ours.[3]

Implementation remark. A practical bottleneck found when running experiments on the first versions of the construction above was the space reserved for Elias codes. The estimated worst case space is $O(n)$ bits but this rarely happens on practical inputs. We chose to reserve initially $o(n)$ bits and double the space if necessary. The parameters were chosen so that the doubling does not affect the overall $O(n \log n)$ worst case time requirement. This reduced the maximum space usage during the construction on common inputs significantly.

6 Lowest Common Ancestor Structure

Farach-Colton and Bender [8] describe a $O(n \log n)$ bits structure that can be preprocessed for a tree in $O(n)$ time to support constant time lowest common

[3] The handling of P is not quite complete in [15]: in some extreme cases, their algorithm might need $O(n \log n)$ bits space. This can be fixed by adding a similar handling of node depths as in our algorithm (their algorithm already has very similar handling of string depths). Alternatively, Hon [14, page 59] describes another solution that goes around this problem.

ancestor (lca) queries. Sadakane [25] modified this structure to take $O(n)$ bits of space without affecting the time requirements. We implemented Sadakane's proposal that builds on top of the balanced parentheses representation of previous section, adding lookup tables taking $o(n)$ bits.

Implementation remark. While implementing Sadakane's proposal, we faced a practical problem; one of the sublinear structures for lca-queries takes space $n(\log\log n)^2/\log n$ bits, which on practical inputs is considerable amount: This lookup table was taking half the size of the complete compressed suffix tree on some inputs. To go around this bottleneck, we added a space-time tradeoff parameter K such that using space $n(\log\log n)^2/(K\log n)$ bits for this structure, one can answer lca-queries in time $O(K)$.

7 Experimental Results

We report some illustrative experimental results on a 50 MB DNA sequence [4]. We used a version of the compressed suffix tree `CST` whose theoretical space requirement is $nH_0 + 10n + o(n\log|\Sigma|)$ bits; other variants are possible by adjusting the space/time tradeoff parameters. Here $n(H_0+1)(1+o(1))+3n$ comes from the compressed suffix array `CSA`, and $6n + o(n)$ from the other structures. The maximum average slowdown on suffix tree operations is $O(\log n\log|\Sigma|)$ under this tradeoff. The experiments were run on a 2.6GHz Pentium 4 machine with 1GB of main memory. Programs were compiled using g++ (GCC) compiler version 4.1.1 20060525 (Red Hat 4.1.1-1) and -O3 optimization parameters.

We compared the space usage against classical text indexes: a standard pointer-based implementation of suffix trees `ST`, and a standard suffix array `SA` were used. We also compared to the *enhanced suffix array* `ESA` [1]; we used the implementation that is plugged into the `Vmatch` software package[5]. For suffix array construction, we used the `bpr` algorithm [26] that is the currently the fastest construction algorithm in practice.

Figure 1 reports the space requirements on varying length prefixes of the text. One can see that the achieved space-requirement is attractive; `CST` takes less space than a plain suffix array.

We also measured the maximum space usage for `CSA` and `CST` during the construction. These values (`CSA, max` and `CST, max`) are quite satisfactory; the maximum space needed during construction is only 1.4 times larger than the final space.

For the time requirement comparison, we measured both the construction time and the usage time (see Fig. 2). For the latter, we implemented a well-known solution to the *longest common substring (LCSS)* problem using both the classical suffix tree and the compressed suffix tree. For sanity check, we also implemented an $O(n^3)$ ($O(n^2)$ expected case) brute-force algorithm.

[4] `http://pizzachili.dcc.uchile.cl/texts/dna/dna.50MB.gz`

[5] http://www.vmatch.de

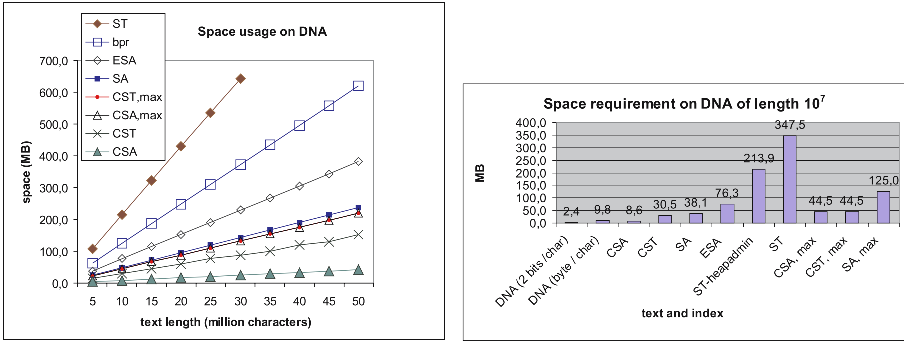


Fig. 1. Comparison of space requirements. We have added the text size to the `SA` and `ST` sizes, as they need the text to function as indexes, whereas `CSA` and `CST` work without. Here `ST-heapadmin` is the space used by suffix tree without the overhead of heap; this large overhead is caused due to the allocation of many small memory fragments. For other indexes, the heap overhead is negligible. Three last values on the right report the maximum space usage during the construction (for `ESA` and `ST` the maximum is the same as the final space requirement).

Table 1. Average running times (in microseconds) for operations of `ST` and `CST`

tree operation	edge(*,1)	sl()	isleaf()	parent()	depth()	lca()
ST	0, 14	0, 09	0, 09	-	-	-
CST	13, 12	11, 07	0, 05	0, 11	4, 56	6, 66

The LCSS problem asks to find the longest substring C shared by two given input strings A and B. The solution using suffix tree is evident: Construct the suffix tree of the concatenation $A\$B$, search for the node whose string depth is largest and its subtree contains both a suffix from A and from B. Notice, that no efficient solution without using suffix tree -alike data structures is known.

Finally, to get an idea how much different types of algorithms will slow down when using the `CST` instead of `ST`, we measured the average execution times of some key operations. We used the DNA sequence prefix of length 5 million for the experiment and ran each operation repeatedly over the nodes of `ST` and `CST`, respectively, to obtain reliable average running time per operation. The results are shown in Table 1.

Notice that `ST` does not support $parent()$, $depth()$, and $lca()$ functions. Such functionalities are often assumed in algorithms based on suffix trees. They could be be added to the classical suffix tree as well (two first easily), but this would again increase the space requirement considerably. That is, the space reduction may in practical settings be even more than what is shown in Fig. 1.

These experiments show that even though the compressed suffix tree is significantly slower than a classical suffix tree, it has an important application domain on genome-scale analysis tasks; when memory is the bottleneck for using classical suffix trees and brute-force solutions are too slow, compressed suffix trees

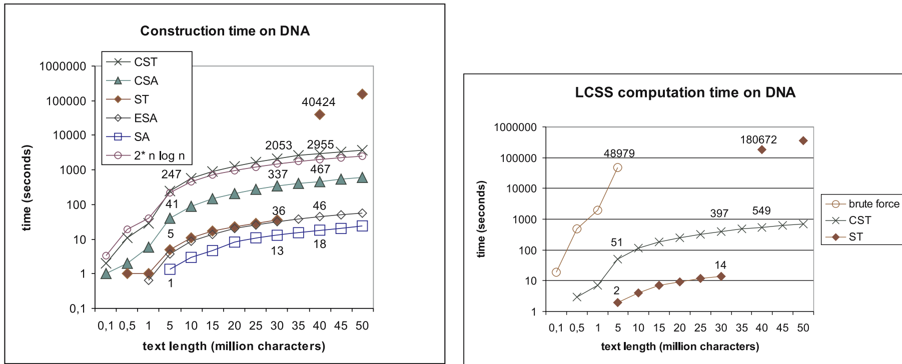


Fig. 2. Comparison of time requirements. For LCSS, we treated the first half of the sequence as A, the second as B. We plotted the expected behaviour, $2n \log n$, for reference. The more dense sampling of x-values is to illustrate the brute-force algorithm behaviour. After 30MB, suffix tree did not fit into main memory. This constitutes a huge slowdown because of swapping to disk.

can provide a new opportunity to solve the problem at hand without running out of space or time. [6]

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2, 53–86 (2004)
2. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words, NATO ISI Series pp. 85–96. Springer, Heidelberg (1985)
3. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report Technical Report 124, Digital Equipment Corporation (1994)
4. Chan, W.-L., Hon, W.-K., Lam, T.-W.: Compressed index for a dynamic collection of texts. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 445–456. Springer, Heidelberg (2004)
5. Cheung, C.-F., Yu, J.X., Lu, H.: Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Transactions on Knowledge and Data Engineering 17(1), 90–105 (2005)
6. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific, Singapore (2002)
7. Elias, P.: Universal codeword sets and representation of the integers. IEEE Transactions on Information Theory 21(2), 194–200 (1975)
8. Farach-Colton, M., Bender, M.A.: The lca problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)

[6] Notice that our suffix tree implementation is not tuned for secondary memory use. When such tuning is done, the slowdown gets very close to that of compressed suffix trees in main memory [5]. Comparison between these two approaches is left for future work.

9. Ferragina, P., Manzini, G.: Indexing compressed texts. Journal of the ACM 52(4), 552–581 (2005)
10. González, R., Grabowski, Sz., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 27–38. Springer, Heidelberg (2005)
11. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. SODA'03, pp. 841–850 (2003)
12. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing 35(2), 378–407 (2006)
13. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
14. Hon, W.-K.: On the Construction and Application of Compressed Text Indexes. PhD thesis, University of Hong Kong (2004)
15. Hon, W.-K., Sadakane, K.: Space-economical algorithms for finding maximal unique matches. In: Apostolico, A., Takeda, M. (eds.) CPM 2002. LNCS, vol. 2373, pp. 144–152. Springer, Heidelberg (2002)
16. Hon, W.-K., Sadakane, K., Sung, W.-K.: Breaking a time-and-space barrier in constructing full-text indices. In: Proc. FOCS'03, p. 251 (2003)
17. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
18. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic Journal of Computing 12(1), 40–66 (2005)
19. Mäkinen, V., Navarro, G.: Dynamic entropy compressed sequences and full-text indexes. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 306–317. Springer, Heidelberg (2006)
20. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing, pp. 935–948 (1993)
21. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96). LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
22. Munro, I., Raman, V., Rao, S.: Space efficient suffix trees. Journal of Algorithms 39(2), 205–222 (2001)
23. Navarro, G.: Indexing text using the Ziv-Lempel trie. Journal of Discrete Algorithms (JDA) 2(1), 87–114 (2004)
24. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys, 2007. To appear, preliminary version available at `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survcompr2.ps.gz`
25. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems, 2007. To appear, preliminary version available at `http://tcslab.csce.kyushu-u.ac.jp/~sada/papers/cst.ps`
26. Schürmann, K.-B., Stoye, J.: An incomplex algorithm for fast suffix array construction. In: Proc. ALENEX/ANALCO, pp. 77–85 (2005)

Simple Space-Time Trade-Offs for AESA

Karina Figueroa[1] and Kimmo Fredriksson[2,⋆]

[1] Facultad de Ciencias Físico-Matemáticas, Universidad Michoacana Mexico
karina@fismat.umich.mx
[2] Department of Computer Science, University of Joensuu
PO Box 111, FIN–80101 Joensuu, Finland
kfredrik@cs.joensuu.fi

Abstract. We consider indexing and range searching in metric spaces. The best method known is AESA, in practice requiring the fewest number of distance evaluations to answer range queries. The problem with AESA is its space complexity, requiring storage for $\Theta(n^2)$ distance values to index n objects. We give several methods to reduce this cost. The main observation is that exact distance values are not needed, but lower and upper bounds suffice. The simplest of our methods need only $\Theta(n^2)$ *bits* (as opposed to words) of storage, but the price to pay is more distance evaluations, the exact cost depending on the dimension, as compared to AESA. To reduce this efficiency gap we extend our method to use b distance bounds, requiring $\Theta(n^2 \log_2(b))$ bits of storage. The scheme uses also $\Theta(b)$ or $\Theta(bn)$ words of auxiliary space. We experimentally show that using $b \in \{1, \ldots, 16\}$ (depending on the problem instance) gives good results. Our preprocessing and side computation costs are the same as for AESA. We propose several improvements, achieving e.g. $O(n^{1+\alpha})$ construction cost for some $0 < \alpha < 1$, and a variant using even less space.

1 Introduction

Similarity searching has a vast number of applications in numerous fields, such as audio, image and document databases, computational biology, and data mining, to name a few. In almost all the applications we have a database of objects and a metric distance function defined between any two objects. Metric space indexing then means preprocessing the database so that subsequent queries can be efficiently answered without comparing the query against the whole database. The most fundamental type of query is *range query*: retrieve all objects in the database that are within a certain similarity threshold to the given query object. A large number of different data structures and query algorithms have been proposed [2,7].

More precisely, we have a universe $\mathbb{U}$ of *objects*, and a non-negative distance function $d : \mathbb{U} \times \mathbb{U} \to \mathbb{R}^+$. The distance function is metric, if it satisfies for all $x, y, z \in \mathbb{U}$

⋆ Supported by the Academy of Finland, grant 207022.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 229–241, 2007.
© Springer-Verlag Berlin Heidelberg 2007

$$d(x,y) = 0 \quad \Leftrightarrow \quad x = y$$
$$d(x,y) = d(y,x)$$
$$d(x,y) \leq d(x,z) + d(z,y).$$

The last item is called the "triangle inequality", and is the most important property in our case as we see later. The database S is a finite subset of that universe, i.e. $S \subseteq \mathbb{U}$. The size of S is $|S| = n$. The database S is preprocessed in order to efficiently answer *range queries*. Given a query object q, we retrieve all objects in S that are close enough to q, i.e. we retrieve the set $\{u \in S \mid d(q,u) \leq r\}$ for some user supplied r. The trivial method is then to directly compute the n distances and return objects that satisfy the condition.

In general metric spaces the (black-box) distance function is the only way to distinguish between the objects. Moreover, the distance function is often very expensive to evaluate (consider e.g. comparing documents or images). Hence the usual complexity measure for range searching is the number of distance function evaluations required to answer the query. In this respect AESA (Approximating Eliminating Search Algorithm) [12] is the baseline. The main problem of AESA is that it requires $\Theta(n^2)$ space to index a database of n objects. Thus a large body of research have aimed to approach the performance of AESA while keeping the space complexity linear [2,7]. All these (as well as AESA) are based on the triangle inequality to discard elements without to compare against the query. However, the linear size index structures are not competitive against AESA. The space complexity of AESA comes from storing a matrix of all the $n(n-1)/2$ pairwise distances between the database objects. The availability of all the distances in constant time makes AESA so powerful. Consequently, several indexing techniques have been developed that try to mimic AESA while using less memory, see Sec. 2. In this paper we present a simple alternative to AESA.

2 Previous Work

AESA [12] and its variants are based on the following fact:

$$|d(q,p) - d(p,o_i)| \ \leq \ d(q,o_i), \tag{1}$$

where q (query), p (pivot, selected object) and o_i (object) are any objects in the universe $\mathbb{U}$. Moreover, p and o_i are also in the database S. Given a range r and a query object q, the task is to retrieve all objects $o_i \in S$ such that $d(q,o_i) \leq r$. In AESA all the $n(n-1)/2$ distances of the form $d(p,o_i)$ are precomputed and stored. The search algorithm then evaluates the distance $d(q,p)$ for some p in the database. If $d(p,q) \leq r$, then p is reported to belong to the range. Then, every object $o_i \in S$ that satisfies

$$|d(q,p) - d(p,o_i)| \ > \ r \tag{2}$$

can be eliminated, since by Eq. (1) it cannot be in the range. In other words, we compute a new set $S' = \{o_i \in S \setminus \{p\} \mid d(q,p) - r \leq d(o_i,p) \leq d(q,p) + r\}$

The distances $d(p, o_i)$ are retrieved from the precomputed matrix. However, the elimination process has to make a linear scan over the set S', so the cost is the time for one distance computation plus $O(n)$. This process is repeated with a new pivot p taken from the qualifying set S', until S' becomes empty. The next pivot can be selected in many ways, picking it at random being the simplest strategy. However, it is better to select it as the object whose lower bound distance to the query is the smallest. These lower bounds can be maintained in $O(n)$ time. Better, but more costly strategies can be found in [3].

LAESA [8] stores only k rows of the distance matrix, hence needing only $\Theta(kn)$ space. The search strategy is basically the same as in AESA. The only difference is that as the number of pivot objects is limited to k, it may not be a good idea to eliminate them in early stages of the search, since they could be used to eliminate other objects later. The preprocessing stage has the additional complexity of deciding which objects to select for pivots. A good strategy is to select objects that are maximally separated [8]. Finally, the parameter k should be as large as possible within the memory constrains to better approximate AESA. The classic pivot-based algorithm is similar to AESA and LAESA, but in this case the pivots are never eliminated during the filtering phase.

More recently, t–spanner graphs have been proposed to approximate the distance matrix [9]. The idea is to explicitly store only some of the distances, while the rest can be approximated within a factor t. The larger t is, the less real distances have to be stored, but the search becomes more costly as less objects can be directly filtered out. The method is also substantially more complex than AESA.

In this article we present a technique that is simpler than t–spanner AESA, and we also have lower cost side computations.

3 Partitioning AESA

The simplest way to reduce the space complexity of AESA is to divide the database into P blocks, each of size n/P objects. One can then build P AESA matrices, requiring a total of $\Theta(n^2/P)$ space (and distance computations). The time trade-off is that to answer the queries, we must run AESA P times. However, for reasonably small P this might still be competitive against many algorithms that take only $\Theta(n)$ space. We call this method PAESA. Similar technique was used in [4], except that a (linear space) index was built over the P blocks.

Therefore, our goal is to reduce the space usage of AESA by some factor of P, while trying to keep the search performance better than P times the cost of AESA.

4 BAESA

We now introduce our method which we will call BAESA, for Binary/Bounded AESA. For each object o_i in our database, we compute and store b distance

bounds, or radii, that is, we build $R_{0\ldots n-1,0\ldots b-1}$. These radii are sorted to increasing order, i.e. $R_{i,h} < R_{i,h+1}$. Moreover, we require that $R_{i,b-1} \geq d(o_i, o_j)$ for any j. Alternatively, we may assume that $R_{i,b-1} = \infty$, and not store it explicitly. The space required for R is nb distances, where each distance may take e.g. one computer word of storage (e.g. 32 or 64 bits). In our case b will be a small number, in the range $2 \ldots 16$, and hence this space is negligible as compared to AESA. Note also that R can replace the original AESA distance matrix if we use $b = n$, and define $R_{i,j} = d(o_i, o_j)$.

It is also possible to use just b radii, as our method works even if $R_{i,h} = R_{j,h}$ for $i \neq j$, i.e. we can use a table $R'_{0\ldots b-1}$ instead. In this case the space required is just b distance values. We consider selecting b and the radii $R_{i,h}$ later.

We also build a *bounded distance index matrix* M, which is defined as

$$M_{i,j} = \min h \mid d(o_i, o_j) \leq R_{i,h}. \tag{3}$$

Note that each entry $M_{i,j}$ takes only $\lceil \log_2(b) \rceil$ *bits* of storage, i.e. a total of $\Theta(n^2 \log(b))$ bits in additional to the space required by R or R'.

The following Lemma (see e.g. [7, Lemma 4.2]) immediately suggests how we can use M and R:

Lemma 1. *Assuming that $lb \leq d(p, o) \leq ub$, it holds that*

$$\max\{d(q, p) - ub, lb - d(q, p)\} \ \leq \ d(q, o).$$

(It also holds that $d(q, o) \ \leq \ d(q, p) + ub$, but this is not interesting for us.)

Our search algorithm is basically the same as in AESA, the only difference is that we do not know the exact distance $d(p, o)$, as this is not stored in the matrix as in AESA. However, we have effectively stored lb (lower-bound) and ub (upper-bound) distances for it. More precisely, assume that we are interested in $d(o_i, o_j)$. Assume that $h = M_{i,j}$. Then we know that

$$lb = R_{i,h-1} \tag{4}$$

$$ub = R_{i,h}. \tag{5}$$

If $R_{i,h-1}$ is not defined, we use simply $lb = 0$. Similarly we can use the distances R'_{h-1} and R'_h. Now, every object $o \in S$ that satisfies

$$\max\{d(q, p) - ub, lb - d(q, p)\} \ > \ r \tag{6}$$

can be eliminated (compare to Eq. (2)), where q is the query object, and r is the range radius.

4.1 LBAESA

Note that we can generalize BAESA in the same way as AESA is generalized to LAESA. That is, instead of using all the n objects in the database as potential pivots, we may choose only k objects. The search algorithm remains essentially the same. In this way, we use an array $R_{0\ldots k-1,0\ldots b-1}$ (or $R'_{0\ldots h-1}$) of distances and matrix $M_{0\ldots k-1,0\ldots n-1}$ of $\lceil \log_2(b) \rceil$ bits per element. Total space then becomes $\Theta(kn \log(b))$ bits and $\Theta(kb)$ words.

4.2 Efficient Construction and Updates

The preprocessing for AESA needs $\Theta(n^2)$ distance computations, to build the distance matrix. Similarly, inserting one new object to a database that contains n objects needs n distance computations. However, deletions are cheap, since that involves only deleting one row and column from the distance matrix. In case of BAESA we must delete one row from R as well.

For BAESA we do not need n distance computations to insert a new object u. Consider the case $b = 2$. We can assign $R_{n,0} = r'$ and $R_{n,1} = \infty$, and use the existing data structure to perform a range query that retrieves all objects with distance at most r' to u, i.e. we compute

$$C = \{o_j \mid d(u, o_j) \leq r',\ o_j \in S,\ 0 \leq j < n\}. \tag{7}$$

This takes only $O(n^\alpha)$ distance evaluations for some $0 < \alpha < 1$, depending on r'. Then we set

$$M_{n,j} = \begin{cases} 0, & o_j \in C, \\ 1, & \text{otherwise}. \end{cases} \tag{8}$$

In fact, we can use this method to build the whole data structure, which then takes only $O(n^{1+\alpha})$ distance computations. In principle this method could be generalized for larger b as well, but the benefits diminish as b and $R_{n,b-2}$ increase.

4.3 Comparison to Other Algorithms

In the following we review two linear space structures and draw parallels to BAESA and BLAESA and show how the algorithms could be combined.

List of Clusters (LC). LC [1] selects a random pivot c, called a *center*, and a *covering radius* $cr(c)$ for the center. The center and its covering radius define a *zone*, and $cr(c)$ is the maximum distance from c to any other object in the zone. A parameter h defines the number of objects in each zone. The list is built as follows. The first center is chosen in random. Then its $h-1$ nearest neighbors are selected, and $cr(c)$ is the distance to the $(h-1)$th neighbor. The zone is then c and its $h-1$ nearest neighbors. The set of these objects is called I, and the rest of the list is recursively built for $E = S \setminus I$. The next center selected is the one that maximizes the sum of distances to all previous centers.

The search evaluates the distance $e = d(q, c)$, and if $e \leq r$ the center c is reported. If $e \leq cr(c) + r$, i.e. the query intersects the zone, the bucket of $h-1$ objects in I is searched exhaustively. If $e > cr(c) - r$, i.e. the query is not fully contained in I, then E (the rest of the list) is searched recursively.

Vantage-Point Tree (VPT). VPT [13] (also known as "Metric Tree" [11]) is basically a balanced binary tree version of LC, up to pivot selection techniques (although historically VPT appeared before LC). That is, I and E contain half of the objects each, and both are built and searched recursively.

Comparison. Consider BLAESA and LC. The latter selects $k = n/h$ pivots (centers), and for each pivot a covering radius is computed. This is in fact precisely the same thing what BLAESA does, in the case of only one bit is used for the distance bound index. The difference is that in the case of BLAESA the covering radius applies to the whole database, not just to the rest of the object list (the E branch) as in LC. The price to pay is that BLAESA needs $\Theta(kn)$ bits in addition to the $\Theta(k)$ radii, but the reward is that at the search phase each "center" can be used to prune objects from *all* the "zones", not only from the current zone as in LC. Hence BLAESA will have better performance.

Similarly, in the same way BAESA would correspond to VPT, if the covering radii is chosen in the same way both in BAESA as in VPT. In this case BAESA would be an improved version of VPT, being able to prune objects from all branches of the tree with each distance evaluation, but again the cost is the additional $\Theta(n^2)$ bits for the matrix M.

Finally, we note that the buckets (I branches) in LC could be implemented with BAESA. The buckets then take $\Theta(h^2)$ bits + $\Theta(h)$ words of space. For small h (so that the size of the matrix M is not dominating the linear space component) this is the same space as the LC would need just to store the list of objects in the bucket, i.e. the search performance of LC can be improved without an increase in space complexity. The same applies to VPT. Similar idea was used in [4], but the buckets were implemented with plain AESA, and hence the space complexity was worse.

5 Compressed Distance Matrix

Even in the best case ($\log_2(b) = 1$) the matrix M takes $\Theta(n^2)$ bits of space. We now show how this can be reduced to $o(n^2)$ bits. The idea is to use compressed dictionaries that provide constant time access to the stored elements [10]. Consider again the case $b = 2$, i.e. each element $M_{i,j}$ is only one bit. We can therefore compress each row (bit-vector) of M using the method that still provides constant time access to $M_{i,j}$ [10]. The number of bits used for a row that has m zero (or one) bits and n bits in total is

$$\log_2 \binom{n}{m} + o(m) + O(\log\log(n)). \tag{9}$$

The first term can be easily shown to be at most the zero-order empirical entropy of the bit sequence, i.e.

$$\log_2 \binom{n}{m} \leq nH_0(M_i) = -m\log_2(m/n) - (n-m)\log_2(1-m/n). \tag{10}$$

The total space then becomes about $n^2 H_0(M) + o(nm)$ bits. Note that we can easily control the value of m, since for each row it is the number of objects that are covered by the radius $R_{j,0}$. For instance, we can choose that radius using the criterion $m = n^\beta$ for some $\beta < 1$, and obtain $o(n^2)$ bits of space.

This idea can be easily generalized for larger b values as well. For example, we can obtain

$$n^2 H_k(M) + o(n^2 \log(b)) \tag{11}$$

bits of space [6] for $k = o(\log_b(n))$, where $H_k(M)$ is the k-order empirical entropy of M. Again, to obtain $H_k(M) < \log_2(b)$ the number of objects covered by the radii must be unbalanced. We note that k-order entropy depends on the order of the objects stored in the database. However, minimizing the entropy for one row (by permuting the columns) of the matrix affects the entropy for the other rows in uncontrollable way. Hence the column permutation should be saved for each row as well, but this would take too much space.

6 Selecting the Radii

Some heuristics to choose the best radii have been proposed: quantiles by element, quantiles by radius, non-uniform partition and using standard deviation. All of them are shown in Fig. 1.

- Quantiles by Element (QBE). A way to select the radii is to divide the set of objects to partitions having equal number of objects. That means that the ith radius is selected so as to cover $i(n/b)$ objects of the set, where b is the number of partitions.
- Quantiles by Radius (QBR). Another way to select the radii is using the histogram of distances and dividing it to equal slices with r. In other words, the ith radius is selected as $r_i = mind + i(maxd - mind)/b$, where $mind$ and $maxd$ are the minimum and maximum pair-wise distances.
- Using standard deviation (SD). This way to select radii is based on standard deviation that is the most common measure of statistical dispersion. We used the values $\pm D$ and $\pm 2D$ and so on around the mean to select the radii, where D is the standard deviation.
- Non-uniform Partition (NUP). In this case we choose the size of every partition in non-uniform way.

Note that since NUP allows us to use any partitioning, all the other methods are special cases of NUP. However, it is not clear what is the optimal way to do the partitioning. In Sec. 7 we study this question experimentaly in the case $b = 2$. The analysis (for a different, linear space data structure) in [1] suggests that the partitioning should be unbalanced, as it is e.g. with QBR.

7 Experimental Results

We made experiments with 3,000 vectors in uniformly distributed unitary cube, since it is a good way to control the dimension of the space, something very difficult in real databases.

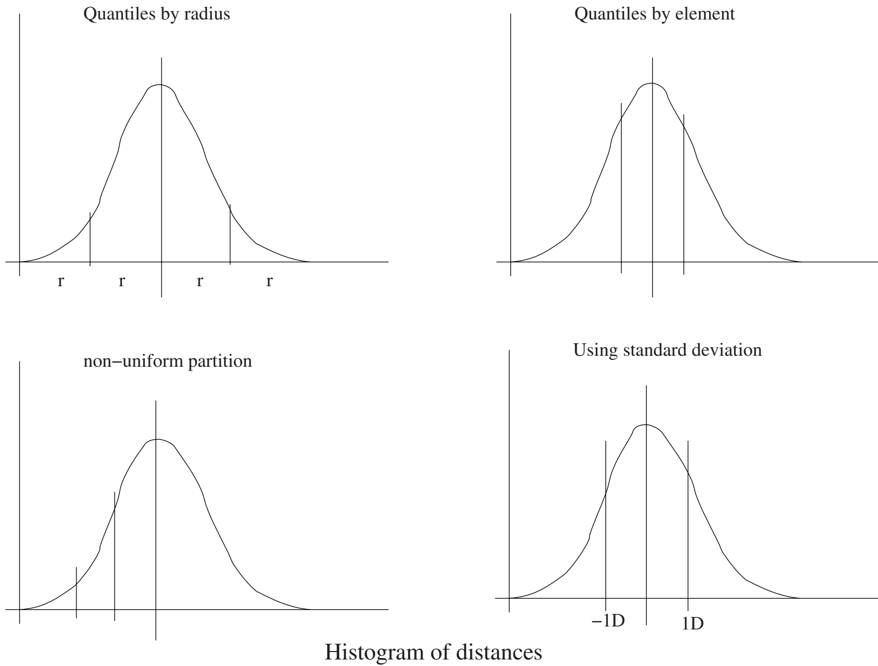


Fig. 1. Criteria to choose the best radii. D is the standard deviation.

BAESA. In Fig. 2 we show the performance of BAESA to retrieve the nearest neighbor in different dimensions. In this plot we use 1 bit for every heuristic to choose the radii, and we compared the performance of a classic pivot-based algorithm using 93 pivot (4 bytes per distance), this is the same amount of memory that we use. Also, it is interesting to compare our performance with a t-spanner. In this case we used t values $1.4 \ldots 1.8$ for dimensions $4 \ldots 20$. Notice that t-spanner use the same amount of memory than our heuristics after dimension 12. In lower dimension t-spanner uses less memory than our heuristics. For NUP we used a radius that covers 10% of the database for dimensions $4 \ldots 10$, and 50% for higher dimensions. Our method uses significantly more distance evaluations than AESA, but we only keep $\Theta(n^2)$ bits + $\Theta(n)$ words against $\Theta(n^2)$ words kept by AESA.

We can improve the performance of BAESA when we use more bits per element. In Fig. 3 we use 4 bits per distance, i.e. 16 distance bounds. This greatly improves the performance. In this plot, again, we use the classic pivot-based algorithm (using 93 and 372 pivots). We also compared the performance of a t-spanner against our algorithms. In this case, with QBE heuristic we used twice the number of distance evaluations as compared to AESA, keeping just $\Theta(n^2 \log(b))$ bits + $\Theta(nb)$ words.

In Fig. 4 we show the performance of BAESA for varying database size, using two different criteria (QBE and QBR) with 1 and 4 bits. By using 4 bits BAESA holds its performance close to AESA even for changes in the size of database.

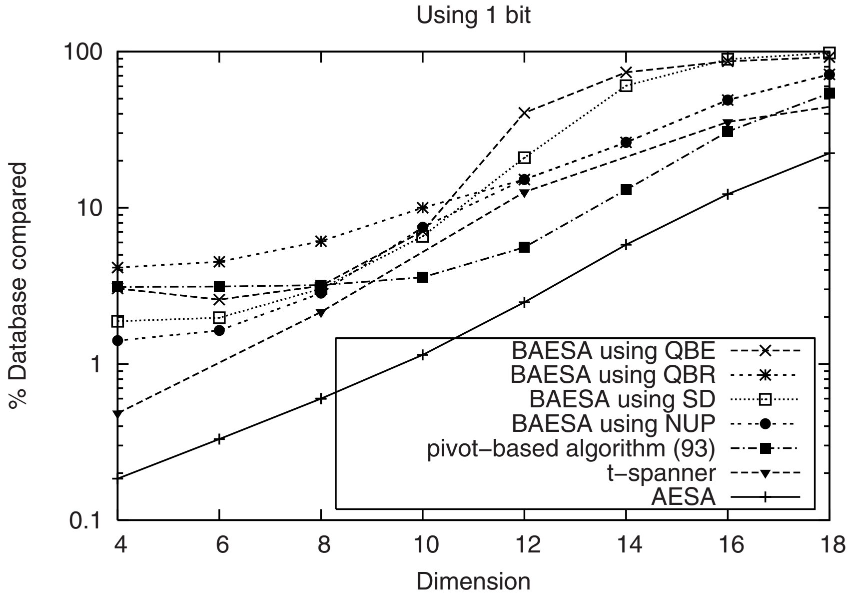


Fig. 2. Comparing AESA, BAESA using the different methods to choose the radii (using 1 bit) and pivot-based algorithm (using 93 pivots)

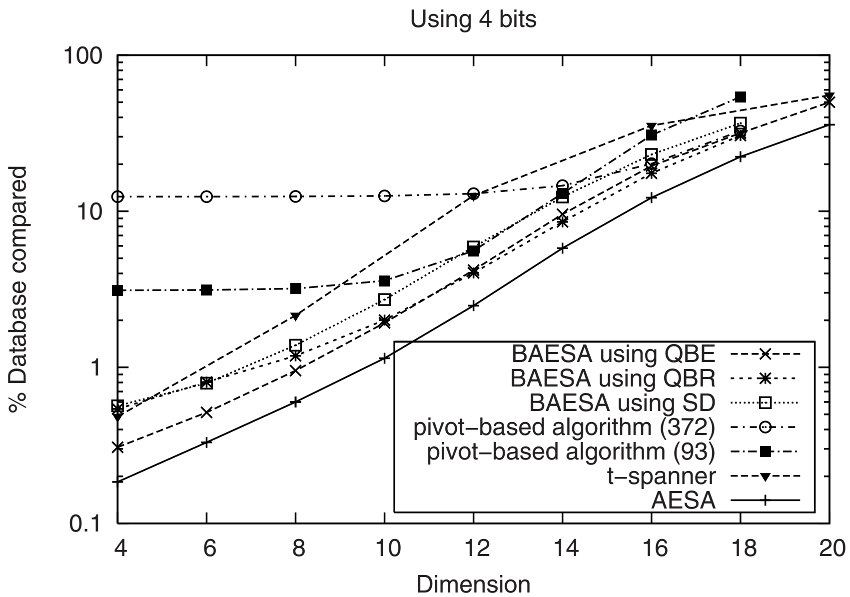


Fig. 3. Comparing AESA, BAESA (using 4 bits) and pivot-based algorithm (using 93 and 372 pivots)

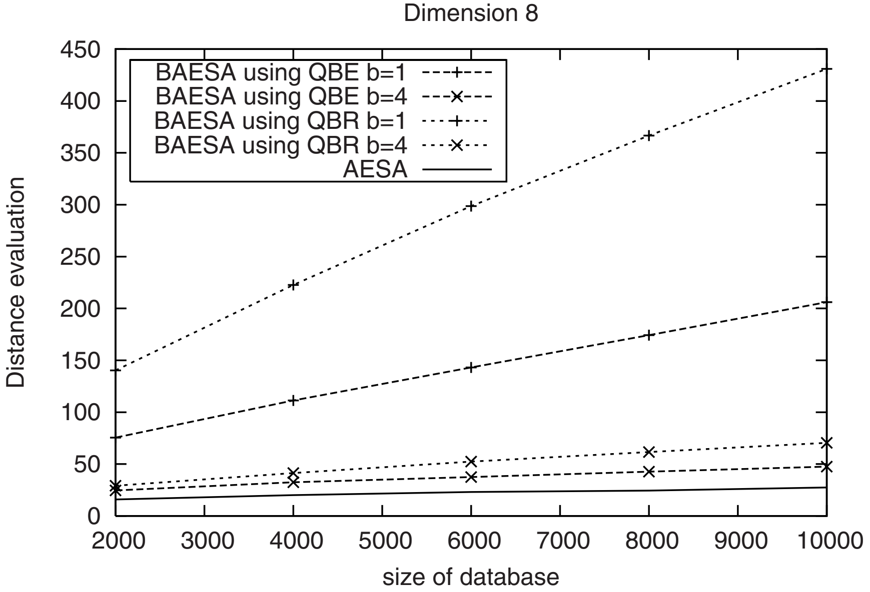


Fig. 4. Comparing AESA and BAESA (using 1 and 4 bits) for different size of the database

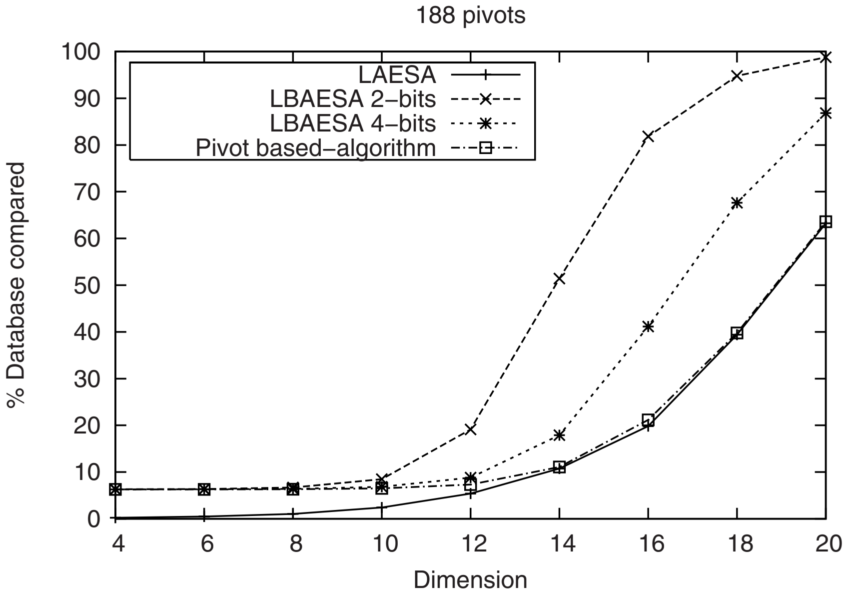


Fig. 5. Comparing pivot-based algorithm, LAESA and LBAESA (using 2 and 4 bits and 188 pivots

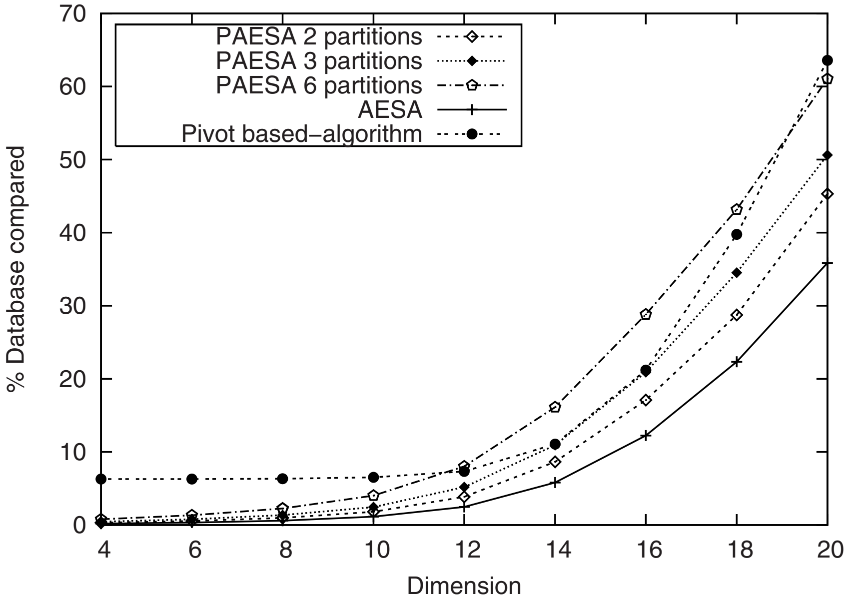


Fig. 6. Comparing pivot-based algorithm (188 pivots), AESA and PAESA (using 2, 3 and 6 partitions) using 3,000 objects

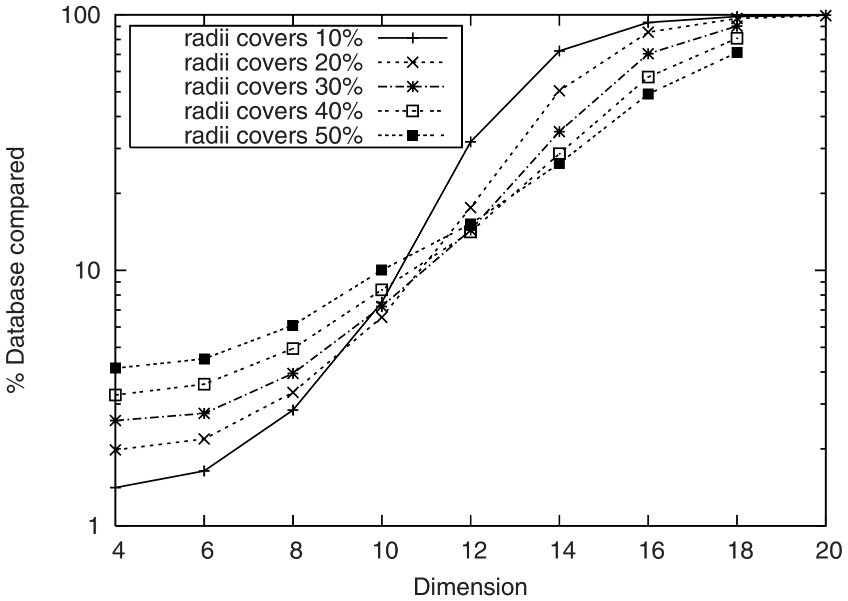


Fig. 7. Changing the values of the first radius. The corresponding $H_0(M)$ values are 0.47 (10%), 0.72 (20%), 0.88 (30%), 0.97 (40%) and 1.00 (50%).

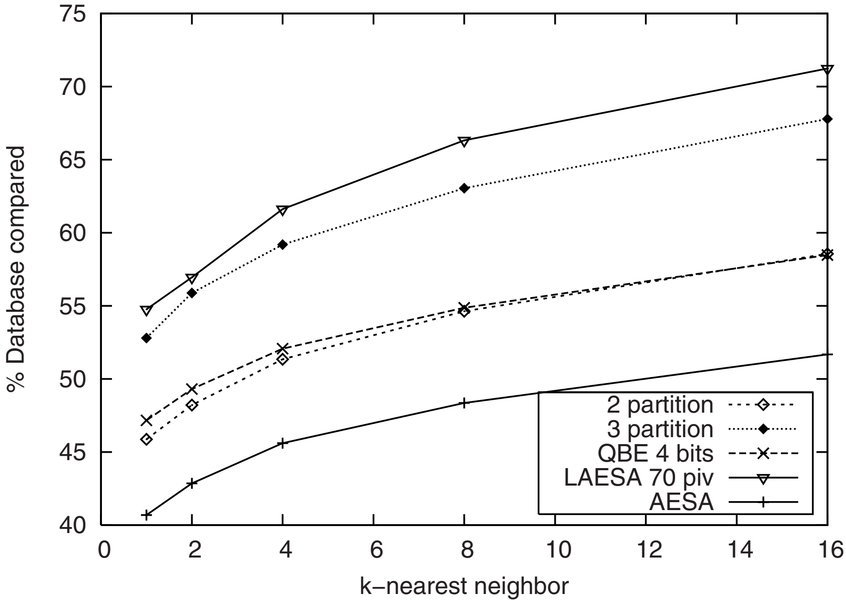


Fig. 8. Comparing our algorithms in a real database (faces): AESA, PAESA (using 2, 3 partitions), QBE (4 bits), LAESA (70 pivots). 1,131 objects.

In Fig. 5 we show the performance of LBAESA using 2 and 4 bits, against pivot-based algorithm using 188 pivots. Notice that 188 pivots used by LAESA and pivot-based algorithm consume much more memory than used by LBAESA, however it is important to show the performance of LBAESA using the same amount of pivots. In this case, after dimension 12 we compared 20% more of the database against LAESA, keeping just $\Theta(kn\log(b))$ bits and $\Theta(kb)$ words against $\Theta(kn)$ words.

Partitioning AESA. The same data (unitary cube) were probed using PAESA and comparing it against pivot-based algorithm using the same amount of memory. The pivots for PAESA were chosen randomly. In Fig. 6 we use 3,000 objects in different dimensions.

Finally, we experimented with BAESA using NUP and only one bit per element, and varying the percentage of the number of objects covered by the first radius. In Fig. 7 we can see that in low dimension a smaller radius is better, but in higher dimension a larger radius is better. Note also that e.g. for a radius covering only 10% of the database the zero-order empirical entropy of the matrix M is about 0.47, which means that we can compress the matrix to about half of its original size.

Real Databases. Finally, we made experiments in a real database of 1,131 faces (CAS-PEAL [5]). The intrinsic dimension of CAS-PEAL is 9. This is a typical database used for pattern recognition. The performance of our algorithms is showed in Fig. 8. As we can see using 2 partitions and QBE with 4 bits we have a good performance in this database. These results are interesting because we

compared only 5% more of the database keeping only half (PAESA) or one eight (QBE) of the memory used by AESA

8 Conclusions

We have proposed a simple variation of AESA. Our method uses significantly less memory than plain AESA, it is very simple to implement and the experimental results are very competitive against previous algorithms that use superlinear space.

Acknowlegments

We are grateful for R. Paredes for his code of t-spanner implementation in order to compare against our algorithms.

References

1. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
2. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Proximity searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)
3. Figueroa, K., Chávez, E., Navarro, G., Paredes, R.: On the lower cost for proximity searching in metric spaces. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 270–290. Springer, Heidelberg (2006)
4. Fredriksson, K.: Engineering efficient metric indexes. Pattern Recognition Letters (PRL) 28(1), 75–84 (2007)
5. Gao, W., Cao, B., Zhou, S., Zhang, X., Zhao, D.: The CAS-PEAL large scale chinese face database and evaluation protocols. Technical report, Join Reserarch & Development Laboratory, CAS, 2004. JDL_TR_04_FR_001
6. González, R., Navarro, G.: Statistical encoding of succinct data structures. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 295–306. Springer, Heidelberg (2006)
7. Hjaltason, G., Samet, H.: Index-driven similarity search in metric spaces. ACM Transactions Database Systems 28(4), 517–580 (2003)
8. Micó, L., Oncina, J., Vidal, E.: A new version of the nearest-neighbor approximating and eliminating search AESA with linear preprocessing-time and memory requirements. Pattern Recognition Letters 15, 9–17 (1994)
9. Navarro, G., Paredes, R., Chávez, E.: t-spanners as a data structure for metric space searching. In: Laender, A.H.F., Oliveira, A.L. (eds.) SPIRE 2002. LNCS, vol. 2476, pp. 298–309. Springer, Heidelberg (2002)
10. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM J. Comput. 31(2), 353–363 (2001)
11. Uhlmann, J.: Satisfying general proximity/similarity queries with metric trees. Information Processing Letters, pp. 175–179 (1991)
12. Vidal, E.: An algorithm for finding nearest neighbors in (approximately) constant average time. Pattern Recognition Letters 4, 145–157 (1986)
13. Yianilos, P.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proc. 4th ACM-SIAM Symposium of Discrete Algorithms (SODA'93), SIAM Press, pp. 311–321 (1993)

Engineering Algorithms for Approximate Weighted Matching *

Jens Maue[1] and Peter Sanders[2]

[1] ETH Zürich, Institute of Theoretical Computer Science, Universitätsstrasse 6, 8092 Zürich, Switzerland
jens.maue@inf.ethz.ch
[2] Universität Karlsruhe (TH), Fakultät für Informatik, Postfach 6980, 76128 Karlsruhe, Germany
sanders@ira.uka.de

Abstract. We present a systematic study of approximation algorithms for the maximum weight matching problem. This includes a new algorithm which provides the simple greedy method with a recent path heuristic. Surprisingly, this quite simple algorithm performs very well, both in terms of running time and solution quality, and, though some other methods have a better theoretical performance, it ranks among the best algorithms.

1 Introduction

Given a graph $G = (V, E)$ with $n := |V|$ nodes and $m := |E|$ edges, a subset of edges $M \subseteq E$ is called a *matching* if no two members of M share an endpoint. A node $u \in V$ is called *matched* if there is an edge $(u, v) \in M$; then, (u, v) is called a *matching* edge and v the *mate* of u. Otherwise, u is called *unmatched* or *free*. If G is weighted and $w\colon E \to \mathbb{R}^{\geq 0}$ denotes the associated weight function, the *weight* of M is defined by $w(M) := \sum_{e \in E} w(e)$, and M is said to be a *maximum weight matching* if there is no matching with larger weight.

The first polynomial time algorithm for the weighted matching problem was given by Edmonds [1] with a running time of $\mathcal{O}(n^2 m)$. This has been improved on repeatedly, and the fastest exact algorithm known so far has an asymptotic running time of $\mathcal{O}(n(m + n \log n))$ for general graphs [2]. Further improvements have been achieved for restricted problems and graph classes such as integer edge weights [3], planar graphs [4], or maximum cardinality matching [5,6].

One of the most recent implementations of the maximum weighted matching problem is that of Mehlhorn and Schäfer [7], which is a variant of the algorithm of Galil, Micali, and Gabow [8]. A history of implementations can be found in [9]. Despite their polynomial running time, these algorithms are too slow for some practical applications on very large graphs or if matchings are computed repeatedly. (Some applications are mentioned in [10].) This motivates the use of approximation algorithms for the maximum weighted matching problem, with a lower—ideally linear—running time that yield very good results.

* Part of this work was done at Max-Planck-Institut für Informatik, Saarbrücken, Germany. Partially supported by DFG grants SA 933/1-2, SA 933/1-3.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 242–255, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Related Work. A well known folklore algorithm achieves a $\frac{1}{2}$-approximation by scanning the edges in descending order of their weights and greedily adding edges between free nodes to the matching [11]. Note that this yields a linear running time for integer edge weights. The first linear time $\frac{1}{2}$-approximation algorithm independent of integer sorting was given by Preis [12]. Later, Drake and Hougardy [13] presented a different $\frac{1}{2}$-approximation, called the Path Growing Algorithm (PGA), whose linear running time is easier to prove. They also developed a $(\frac{2}{3} - \epsilon)$-algorithm [10,14] running in time $\mathcal{O}(\frac{n}{\epsilon})$. A simpler $(\frac{2}{3} - \epsilon)$-approximation algorithm running in time $\mathcal{O}(n \log \frac{1}{\epsilon})$ was later developed by Pettie and Sanders [15]. Several $\frac{1}{2}$-approximation algorithms were evaluated experimentally in [14]. This included a version of PGA improved by a path heuristic not affecting the linear running time; this version of PGA is called PGA$'$. This algorithm performed very well in the experiments, so the greedy algorithm with its super-linear running time was concluded to be a bad choice in most cases.

Our Contribution. As a seemingly trivial yet crucial contribution, we regard the measured solution qualities of the approximation algorithms in relation to the optimal solution. This not only demonstrates that these algorithms achieve solutions of a quality much better than their worst case guarantees suggest, it also makes comparing the algorithms with each other more meaningful. Although the solution quality of the algorithms differ by only a few percent, the *gap to optimality* can differ by a large factor. We also give the first experiments for real-world inputs from an application that is often cited as a main motivation for applying approximation algorithms for weighted matching.

Our most important algorithmic contribution is a rehabilitation of the greedy approach through a new algorithm called GPA, which is described in Sect. 2 in detail. Basically, GPA applies the earlier mentioned path heuristic used for PGA$'$ [14] to the greedy algorithm, which makes the previously outclassed greedy method jump ahead to one of the best practical algorithms. In particular, GPA is usually *faster* than the randomized $(\frac{2}{3} - \epsilon)$-approximation algorithm (RAMA) from [15] and often outperforms it in terms of solution quality. The overall winner is another new algorithm that first executes GPA and then applies a modified version of RAMA, called ROMA. This combination is as fast as ROMA alone since GPA accelerates the convergence of ROMA.

Outline. The tested approximation algorithms are described in Sect. 2. Section 3 introduces the graph instances, on which the tests are performed as presented in Sect. 4. The main results are summarized in Sect. 5.

2 Algorithms

Greedy. The well-known algorithm shown in Fig. 1 follows a simple greedy strategy [11]: repeatedly, the currently heaviest non-matching edge with free endpoints is added to the matching until no edges are left.

```
GRDY(G = (V, E), w : E → ℝ^{≥0})
1 M := ∅
2 while E ≠ ∅ do
3    let e be the edge with biggest weight in E
4    add e to M
5    remove e and all edges adjacent to its endpoints from E
6 return M
```

Fig. 1. The greedy algorithm for approximate weighted matchings

The greedy algorithm runs in time $\mathcal{O}(m + \text{sort}(m))$, where $\text{sort}(m)$ denotes the time for sorting m items. This yields a linear running time for integer edge weights and $\mathcal{O}(m \log n)$ for comparison-based sorting. Let $M^\star$ be a maximum weight matching and M a matching found by the greedy algorithm. Every time an edge e is added to M, at most two edges $e_1, e_2 \in M^\star$ are removed from the graph. Since both $w(e) \geq w(e_1)$ and $w(e) \geq w(e_2)$, the greedy matching M satisfies $2w(M) \geq w(M^\star)$, so the performance ratio is $\frac{1}{2}$.

```
PGA'(G = (V, E), w : E → ℝ^{≥0})
 1 M := ∅
 2 while E ≠ ∅ do
 3    P := ⟨⟩
 4    arbitrarily choose v ∈ V with deg(v) > 0
 5    while deg(v) > 0 do
 6       let e = (v, u) be the heaviest edge adjacent to v
 7       append e to P
 8       remove v and its adjacent edges from G
 9       v := u
10    M := M ∪ MaxWeightMatching(P)
11 extend M to a maximal matching
12 return M
```

Fig. 2. The improved Path Growing Algorithm PGA′

PGA′. The improved version of the Path Growing Algorithm by Drake and Hougardy is referred to by PGA′ [14]. The original algorithm (PGA) [13] without improvements first grows a set of node disjoint paths one after another. Each path is built starting from an arbitrary free node, repeatedly extending it along the heaviest edge $e = (u, v)$ adjacent to its current endpoint u with a free opposite endpoint v and deleting all edges adjacent to u. PGA starts a new path if the current one cannot be extended and finishes when no edges are left. While growing the paths, selected edges are alternately added to two different matchings M_1 and M_2, and the heavier one is finally returned.

The algorithm processes each edge at most once and thus has a linear running time of $\mathcal{O}(m)$. The algorithm yields an approximation ratio of $\frac{1}{2}$, which can be shown by assigning every edge to the end node from which it was deleted during a run of the algorithm. Then, for every edge e of a maximum weight matching M, there is an edge $e' \in M_1 \cup M_2$ with $w(e') \geq w(e)$ which is adjacent to the node that e is assigned to. Therefore, $w(M_1 \cup M_2) \geq w(M)$ and the heavier set of M_1 and M_2 has a weight of at least $\frac{1}{2}w(M)$.

The improved version (PGA$'$) of this algorithm is shown in Fig. 2. Instead of alternately adding the edges of a path to the two matchings, PGA$'$ calculates an optimal matching for every path. This process does not affect the asymptotic running time since computing a maximum weight matching of a path by dynamic programming requires a linear time in the length of the path, which is described at the end of this section. Furthermore, the contribution of a path to the final matching can only increase in weight compared to the original algorithm, so the approximation ratio of $\frac{1}{2}$ is not impaired. As the second improvement, the computed matching is extended to a maximal matching at the end of PGA$'$, which is done by just scanning all edges once without increasing the running time of $\mathcal{O}(m)$.

GPA($G = (V, E)$, $w : E \to \mathbb{R}^{\geq 0}$)

1 $M := \emptyset$
2 $E' := \emptyset$
3 **for each** edge $e \in E$ in descending order of their weight **do**
4   **if** e is applicable **then** add e to E'
5 **for each** path or cycle P in E' **do**
6     $M' :=$ MaxWeightMatching(P)
7     $M := M \cup M'$
8 **return** M

Fig. 3. The Global Paths Algorithm GPA

GPA. Our Global Paths Algorithm (GPA) shown in Fig. 3 presents a new approximation method by integrating the greedy algorithm and PGA$'$. GPA generates a maximal weight set of paths and even length cycles and then calculates a maximum weight matching for each of them by dynamic programming. These paths initially contains no edges and hence represent n trivial paths—isolated nodes. The set is then extended by successively adding applicable edges in descending order of their weight. An edge is called *applicable* if it connects two endpoints of different paths or the two endpoints of an odd length path. An edge is not applicable if it closes an odd length cycle, or if it is incident to an inner node of an existing path. Once all edges are scanned, a maximum weight matching is calculated for each path and cycle.

The test whether an edge is applicable can be done in constant time, and growing the set of paths and cycles needs a linear number of such tests. Calculating a maximum weight matching for a given path needs linear time in the

length of the path using the dynamic programming approach already mentioned above and described below. Thus, GPA needs a linear amount of time after the edges have been sorted, so GPA has a running time of $\mathcal{O}(\text{sort}(m) + m)$, which is $\mathcal{O}(m \log n)$ in general.

Let $M^\star$ be an optimum matching and $P \subset E$ the set of edges that are added to some path or cycle in the first round of GPA. For every $e \in M^\star \setminus P$, e was not applicable at the time considered by GPA, so there are two edges $e_1, e_2 \in P \setminus M^\star$ adjacent to e with both $w(e_1) \geq w(e)$ and $w(e_2) \geq w(e)$. Conversely, every member of P is adjacent to at most two edges of $M^\star$, so there is an injective mapping $f : M^\star \to P$ with $w(f(e)) \geq w(e)$, which implies $w(P) \geq w(M^\star)$. Moreover, the maximum weight matching of a path has a weight of at least half the weight of the path—which also holds for even length cycles. Therefore, any matching M computed in the first round of GPA satisfies $w(M) \geq \frac{1}{2}w(P) \geq \frac{1}{2}w(M^\star)$, so GPA has a performace ratio of $\frac{1}{2}$. This ratio is tight as Fig. 4 shows: in the example graph GPA finds a matching with a weight of not more than $\frac{m}{4}(c + \epsilon) = \frac{1}{2}\, w_{\text{opt}} + \epsilon'$.

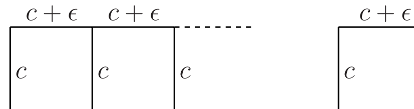


Fig. 4. This graph with a maximum weight matching of weight $w_{\text{opt}} = \frac{m}{2}\, c$ shows that the approximation ratio of $\frac{1}{2}$ is tight for the greedy algorithm, PGA′, and GPA

A second round of GPA can be run on the set of remaining edges with two unmatched endpoints after GPA finishes. We even allow up to three rounds, but the algorithm almost never runs the third round in the experiments presented in Sect. 4 since no applicable edges are left. Running GPA a second time on the result produced by itself presents an alternative to the postprocessing used in PGA′, which extends the computed matching to a maximal matching by simply collecting edges with two free endpoints. The postprocessing of PGA′ could clearly be applied to GPA too.

RAMA. A path P is *alternating* if it consists of edges drawn alternately from M and $E \setminus M$. An alternating path P is called an *augmentation* if the symmetric difference $M \oplus P = (M \setminus P) \cup (P \setminus M)$ is a matching too and $w(M \setminus P) > w(P \setminus M)$. The *gain* of an alternating path P is defined by $g(P) = w(P \setminus M) - w(P \cap M)$. A *k-augmentation* is an augmentation with at most k non-matching edges, and a 2-augmentation P is called *centered* at v if all edges of $P \setminus M$ are incident to either v or its mate. Finally, a maximum-gain 2-augmentation centered at v is denoted by $\text{aug}(v)$.

Figure 5 shows a randomized algorithm by Pettie and Sanders [15]. It repeatedly chooses a random node and augments the current matching with the highest-gain 2-augmentation centered at that node. The way how to find this

```
RAMA(G = (V, E), w: E → IR^{≥0}, int k)
1 M := ∅ (or initialise M with any matching)
2 for i := 1 to k do
3     randomly choose v ∈ V
4     M := M ⊕ aug(v)
5 return M
```

Fig. 5. The Random Augmentation Matching Algorithm RAMA

highest-gain 2-augmentation—according to which the algorithm is implemented—is described in [15]. All this is iterated k times, and by putting $k := \frac{1}{3}\, n \log \frac{1}{\epsilon}$ the algorithm has an expected running time of $O(m \log \frac{1}{\epsilon})$ and an expected performance ratio of $\frac{2}{3} - \epsilon$ [15].

```
ROMA(G = (V, E), w: E → IR^{≥0}, int ℓ)
1 M := ∅ (or initialise M with any matching)
2 for i := 1 to ℓ do
3     for each node v ∈ V in random order do
4         M := M ⊕ aug(v)
5 return M
```

Fig. 6. The Random Order Augmentation Matching Algorithm ROMA

ROMA. The new Random Order Augmentation Matching Algorithm (ROMA) shown in Fig. 6 presents a variant of RAMA from above. The algorithm operates in phases whose number is denoted by ℓ (in Fig. 6, lines 3–4 together form one phase): in every phase, the algorithm successively selects all nodes in random order, and the current matching is repeatedly augmented with the highest-gain 2-augmentation centered at the node currently selected. As described in [15], the time required to find the highest-gain 2-augmentation $\text{aug}(v)$ centered at v is $\mathcal{O}(\deg v + \deg(\text{mate}(v)))$, so the time averaged over all nodes required for one phase of n iterations is $\mathcal{O}(m)$. By setting the number of phases properly, running time and performance ratio of ROMA correspond to those of RAMA.

The randomized algorithms can be initialized with the empty matching or any (non-empty) matching found by one of the algorithms GRDY, PGA′, or GPA. Several combinations are tested in Sect. 4. A similar approach is followed in [14], where a maximal set of node disjoint 2-augmentations is calculated in linear time to improve a previously computed matching.

Maximum Weighted Matching for Paths and Cycles. Some of the algorithms described above compute maximum weighted matchings for paths or cycles, which is done using the dynamic programming algorithm shown in Fig. 7. In the description, $P := \langle e_1, \ldots, e_k \rangle$ denotes a solution of the subproblem for the path $\langle e_1, \ldots, e_i \rangle$, and $W[i]$ denotes the corresponding weight.

```
MaxWeightMatching(P = ⟨e_1, ..., e_k⟩)
1  W[0] := 0;  W[1] := w(e_1)
2  M[0] := ∅;  M[1] := {e_1}
3  for i := 2 to k do
4    if w(e_i) + W[i − 2] > W[i − 1] then
5      W[i] := w(e_i) + W[i − 2]
6      M[i] := M[i − 2] ∪ {e_i}
7    else
8      W[i] := W[i − 1]
9      M[i] := M[i − 1]
10 return M[k]
```

Fig. 7. Obtaining a maximum weight matching for a path by dynamic programming

The algorithm has a running time which is linear in the length of the input path. In order to compute the maximum weight matching of a cycle C, let e_1 and e_2 be any two consecutive edges of C, let P_1, P_2 be the paths obtained by removing e_1, e_2 from C respectively, and let M_1 and M_2 denote the maximum weight matching of P_1 and P_2 respectively. Then, both M_1 and M_2 are valid matchings for C, and the one of larger weight must be a maximum weight matching for C since not both e_1 and e_2 can be member of a maximum weight matching of C. Thus, maximum weight matchings for cycles can be found in linear time of their length too.

3 Test Instances

Three families of (synthetic) instances taken from [7] are presented below, followed by a description of the real-world graphs. We used the C++ library LEDA 4.5 [16] for some steps of the generation process as described below.

Delaunay Instances are created by randomly choosing n points in the unit square and computing their Delaunay triangulation using LEDA. The numbers of points n are chosen to be $n := 2^x$ with $x \in \{10, \ldots, 18\}$, giving nine different Delaunay graphs in total. Their edge weights are obtained by scaling the Euclidean distances in the unit square to integers in the range between 0 and 2^{31-x}.

Complete Geometric Instances are generated each by choosing n random points in an $(n \times n)$-square. n is set to $n := 2^x$, now with $x \in \{6, \ldots, 12\}$, yielding seven instances. The edge weights correspond to the Euclidean distances between their endpoints, rounded off to integer values.

Random Instances are generated with the LEDA implementation for random simple undirected graphs. We generate 64 graphs by choosing eight different values for n and eight different values of m for each n: for every $n := 2^x$, $x \in \{10, \ldots, 17\}$, eight random graphs are created by putting $m := 2^y n$, $y \in \{1, \ldots, 8\}$. The edges are assigned random integer weights between 0 and 2^{31-x}.

Real-World Instances. Some of the best known practical graph partitioning algorithms reduce the size of their input graph by successively contracting them in the followin way: starting with unit edge weights, an approximate weighted matching is computed in each contraction for the current graph using PGA′ [14]. Then, every matching edge is contracted into a single node. Parallel edges resulting from a contraction are replaced by a single edge whose weight is the sum of the weights of its constituent edges. The initial 34 graphs are taken from [17], for each of which eight contractions are successively applied, yielding 272 instances with integer edge weights.

4 Experimental Results

The algorithms described in Sect. 2 are run on the instances presented above and compared by their solution quality and running times. The RAMA and ROMA algorithms are further tested in combination with initial matchings obtained by the other algorithms. The algorithms were implemented using the data structure 'graph' and the sorting algorithms of the C++-library LEDA-4.5 [16] and compiled with the GNU C++-compiler g++-3.2.

All tested algorithms actually perform much better than their worst case bounds suggest, and even the worst approximate solution in the tests has a relative error of less than 10 %. Still, the measured error, i.e. the gap to optimality, differs by a large factor between the best and worst algorithm, and whether some algorithm performs better than another depends on the graph family. In this section, the ratio between the number of nodes and edges of a graph will be denoted by α, i.e. $\alpha := \frac{m}{n}$.

The presented running times should not be over-interpreted since the algorithms could probably be implemented more efficiently. In particular, using fast integer sorting would probably accelerate GPA considerably. Still, we believe that the plotted running times are meaningful to some degree since in a sense all the implementations are "of the same quality" and they all use LEDA.

4.1 Solution Quality and Running Times

Figures 8 and 9 show the results for synthetic instances. For the random graphs Fig. 8 contains two diagrams for each $\alpha \in \{4, 16, 64, 256\}$. The x-axis shows the number of nodes n. The y-axis shows the difference of an algorithm's approximate solution to the optimum in percent (left column) and the corresponding running times (right) respectively. The randomized algorithms RAMA and ROMA perform up to $8n$ augmentation steps. ROMA cancels if its current matching is *saturated*, i.e. if one phase of n iterations has not further improved it.

Figure 8 shows that for random graphs of any value of α, GPA performs significantly better than the greedy algorithm and PGA′. The randomized algorithms RAMA and ROMA produce approximations even better than GPA for $\alpha < 16$, but this advantage decreases with increasing α: their solution quality is almost equal for $\alpha = 16$, whereas RAMA and ROMA are outperformed by

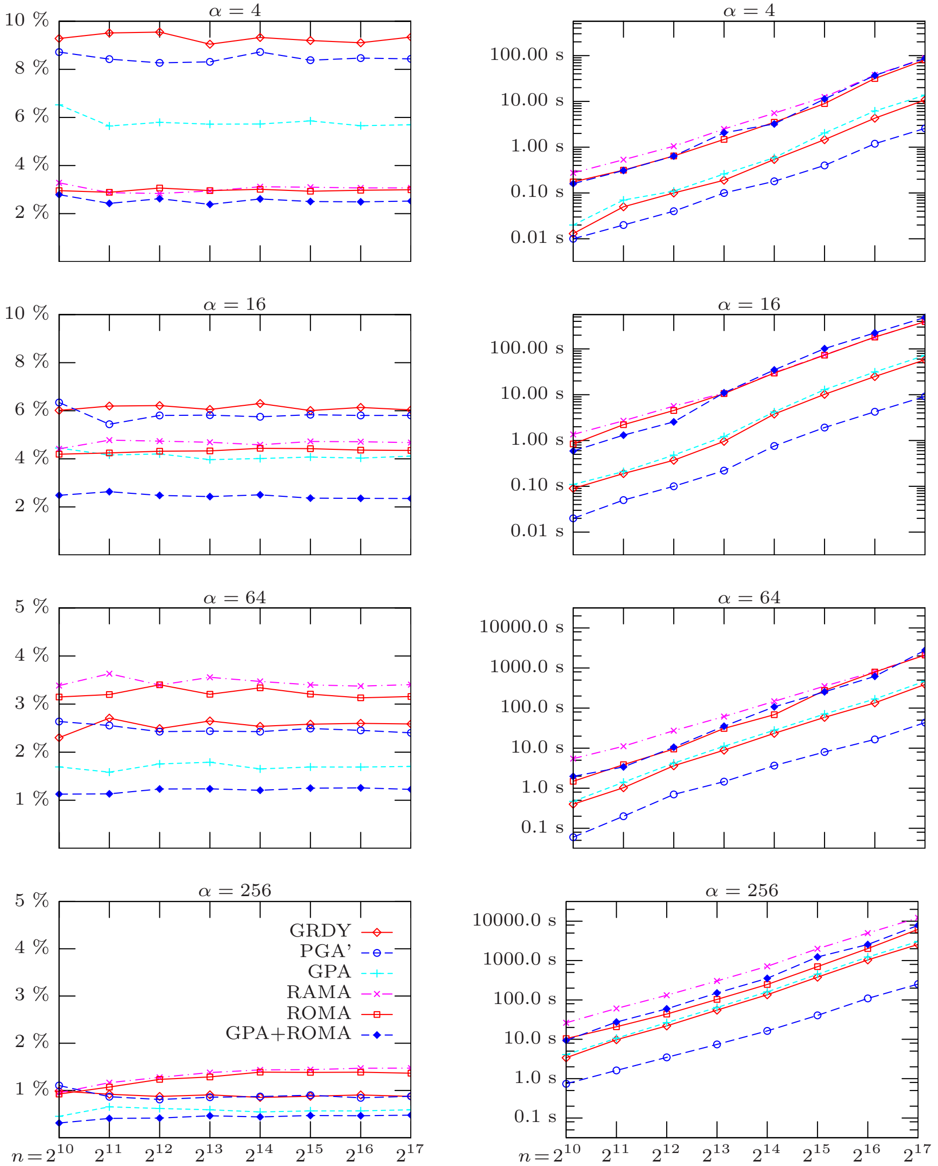


Fig. 8. Gap to optimality (left) and running times (right) for random graphs with $n = 2^{10}, \ldots, 2^{17}$ and $\alpha \in \{4, 16, 64, 256\}$. The key applies to all plots.

GPA for graphs with $\alpha > 16$, and also by the greedy algorithm and PGA$'$ for higher values of α. Moreover, GPA followed by post-processing through ROMA performs best for any value of α. The linear time algorithm PGA$'$ runs fastest in practice. Interestingly, the superlinear time algorithms greedy and GPA are

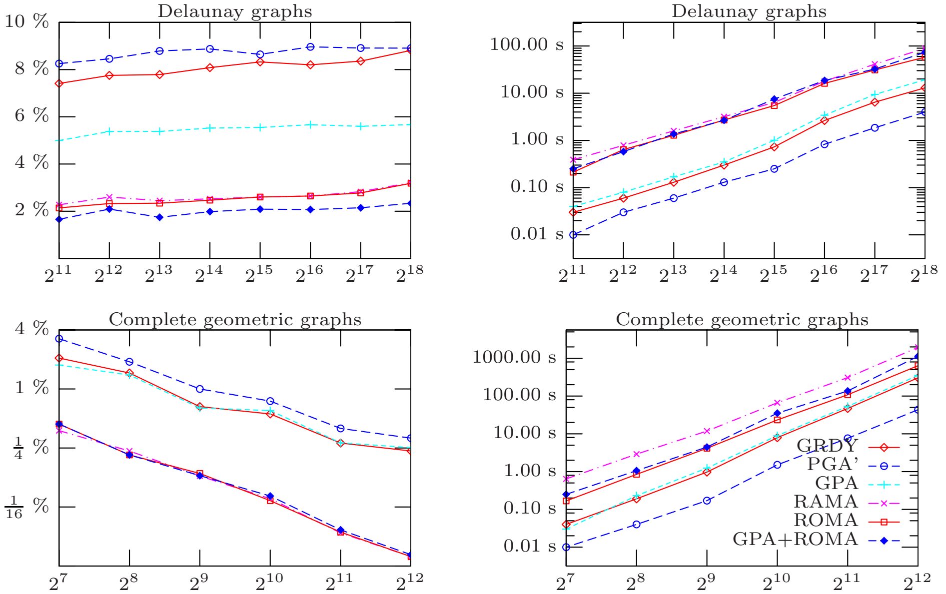


Fig. 9. Gap to optimality (left) and running times (right) for Delaunay (top) and complete geometric (bottom) graphs. The key applies to all plots.

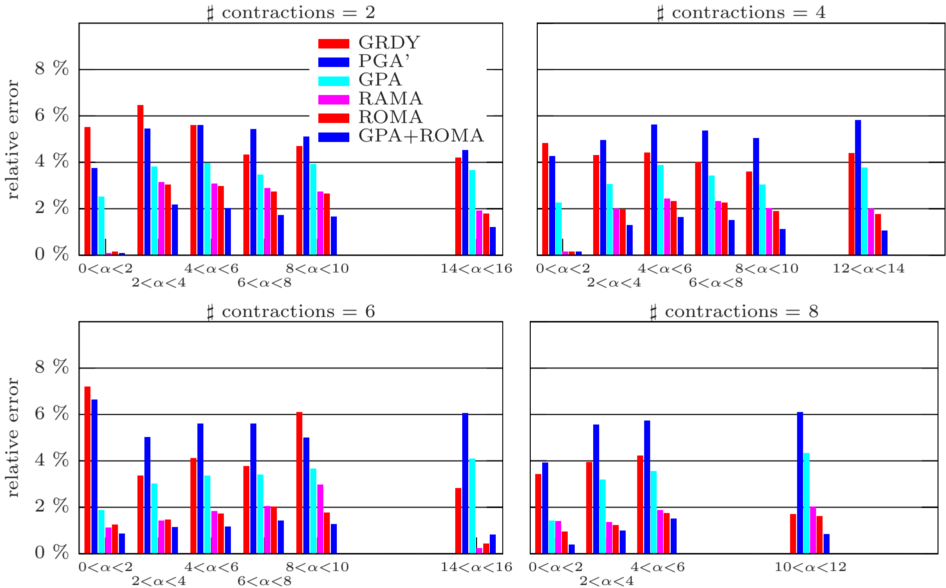


Fig. 10. Gap to optimality for real-world graphs. Each plot corresponds to a class of graphs created by the same number of contractions and shows the average solution quality, where graphs with a similar value of α are combined.

faster than the linear time algorithms RAMA and ROMA. The reason seems to be that the constant factors involved in sorting are much smaller than the constant factors in RAMA and ROMA, which involve multiple passes over the graph involving a massive number of random accesses to the graph data structure that can cause cache faults. ROMA is generally faster than RAMA since it usually finishes earlier due to saturation. Furthermore, calculating an initial matching with GPA causes an even earlier saturation, which makes up for the additional preprocessing time.

The tested Delaunay instances have values $\alpha \approx 3$, and the solution qualities and running times are similar to those for random graphs with similar values of α (see Fig. 9). Also for the complete geometric instances, for which $\alpha = \frac{n-1}{2}$, the algorithms perform just as expected from random graphs of corresponding α. For both instance families the main difference to random graphs is that the greedy algorithm performs better than PGA$'$, but GPA combined with ROMA still produces the best results.

Figure 10 shows the solution quality for the real-world graphs, each plot corresponding to a set of graphs derived by the same number of contractions. The x-axis shows the graphs' values of α grouped by similar values. The y-axis shows the deviation from the optimal solution. The results for real-world graphs basically support what has been concluded for synthetic graphs. In particular, GPA followed by ROMA performs best in almost all cases.

4.2 Convergence of Randomized Algorithms

Figure 11 shows how RAMA and ROMA converge for the random instances. The left column of Fig. 11 contains one diagram for each $\alpha \in \{4, 16, 64, 256\}$, with the x-axis showing the number of iterations and the y-axis showing the relative error (average value over several runs) after this number of steps.

Even with an initially empty matching, i.e. without preprocessing, the randomized algorithms quickly converge on an almost saturated matching with a small error and do not show much improvement after about $4n$ steps for any α. The bigger the value of α, the better the first approximation after n steps is. Used as stand-alone methods, ROMA performs better than RAMA again. Furthermore, starting with an initial matching—obtained by any of the greedy algorithm, PGA$'$, or GPA—yields a significant improvement. GPA clearly achieves the best initialization, which can be improved up to n or $2n$ iterations of ROMA depending on the instance, and the solution quality cannot be achieved with a different preprocessing method using more iterations.

The plotted lines are cut off if no improvement is achieved anymore. Since the errors are averages over several graphs and runs, this only happens if the matching is saturated in ALL sample runs for this number of iterations. However, saturation is achieved earlier on the average, and the right column of Fig. 11 describes after how many phases ROMA achieves saturation on the average.

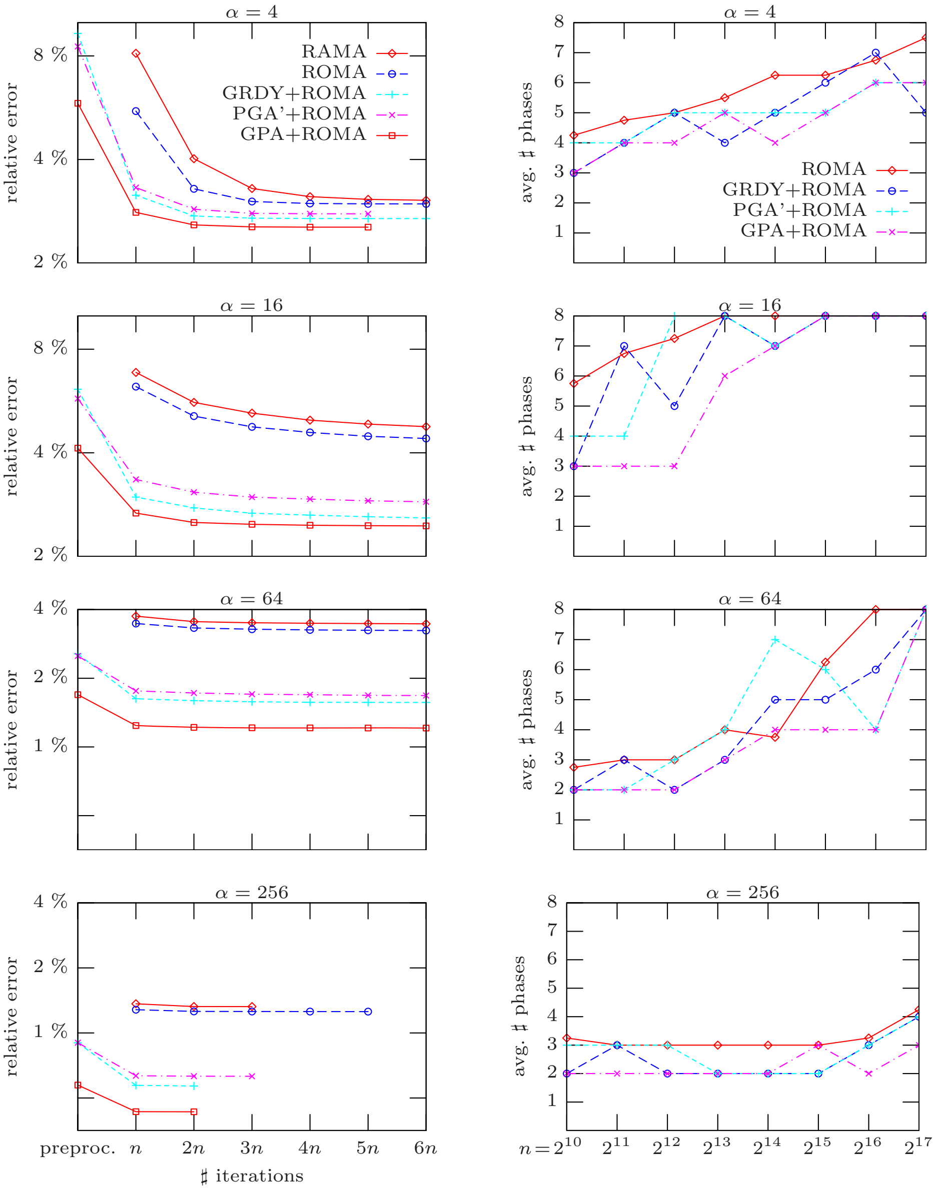


Fig. 11. Convergence (left) and average number of phases until saturation (right) of RAMA and ROMA for the random instances with $\alpha \in \{4, 16, 64, 256\}$

5 Conclusion

The simple greedy algorithm, PGA′, and GPA all produce solutions of a quality much better than their lower bound guarantees. Applying the path heuristic from PGA′ to the greedy algorithm yields a very high improvement: GPA produces

very good solutions at a reasonable running time, so sorting-based matching algorithms remain interesting and must not be neglected.

Moreover, the randomized algorithms also perform very well. As stand-alone methods they are superior in terms of solution quality for graphs of small ratios of $\frac{m}{n}$, and they are highly suitable to post-process existing matchings. The variant ROMA improves RAMA, and its combination with GPA shows the best experimental results among all methods tried.

Future Work. For more meaningful running time comparisons, it would be interesting to compare tuned versions of the best algorithms presented here (GPA, ROMA, and optimal). We see a considerable potential for optimization: GPA does not need complicated graph data structures, and the LEDA function `sort_edges` we currently use, which seems to be quite slow, could be replaced by integer sorting. ROMA (and RAMA) could use static graph data structures optimized for the required operation mix. More interestingly, these algorithm could also be changed in a way that only non-saturated nodes are considered as centers for augmenting paths. The main task here is how to maintain the set of candidate centers efficiently.

Since parallel processing is becoming ubiquitous, we could also consider parallelization. In GPA we can parallelize sorting and independent dynamic programming problems. We can consider possible augmentations for ROMA in parallel, but we have to be careful when actually performing an augmentation.

Another interesting question is whether the running time of exact weighted matching algorithms can be improved by calculating initial matchings with the presented approximation algorithms.

Acknowledgements. We would like to thank Seth Pettie for interesting discussions about the subject. We would also like to thank the reviewers for their detailed comments.

References

1. Edmonds, J.: Maximum matching and a polyhedron with 0, 1-vertices. J. Res. Nat. Bur. Standards 69B, 125–130 (1965)
2. Gabow, H.N.: Data structures for weighted matching and nearest common ancestors with linking. In: Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-90), SIAM, pp. 434–443 (1990)
3. Gabow, H.N., Tarjan, R.E.: Faster scaling algorithms for general graph matching problems. Journal of the ACM 38, 815–853 (1991)
4. Lipton, R.J., Tarjan, R.E.: Applications of a planar separator theorem. SIAM Journal on Computing 9, 615–627 (1980)
5. Micali, S., Vazirani, V.V.: An $\mathcal{O}(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In: Proceedings of the 21st Annual Symposium on Foundations of Computer Science (FOCS-80), pp. 17–27. IEEE Press, New York (1980)
6. Vazirani, V.V.: A theory of alternating paths and blossoms for proving correctness of the $\mathcal{O}(\sqrt{V}E)$ general graph maximum matching algorithm. Combinatorica 14, 71–109 (1994)

7. Mehlhorn, K., Schäfer, G.: Implementation of O(nm log n) weighted matchings in general graphs: The power of data structures. ACM Journal of Experimental Algorithms, vol. 7 (2002)
8. Galil, Z., Micali, S., Gabow, H.N.: An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. SIAM Journal on Computing 15, 120–130 (1986)
9. Cook, W., Rohe, A.: Computing minimum-weight perfect matchings. INFORMS Journal on Computing 11, 138–148 (1999)
10. Drake Vinkemeier, D.E., Hougardy, S.: A linear-time approximation algorithm for weighted matchings in graphs. ACM Trans. Algorithms 1, 107–122 (2005)
11. Avis, D.: A survey of heuristics for the weighted matching problem. Networks 13, 475–493 (1983)
12. Preis, R.: Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In: Meinel, C., Tison, S. (eds.) STACS 99. LNCS, vol. 1563, pp. 259–269. Springer, Heidelberg (1999)
13. Drake, D.E., Hougardy, S.: A simple approximation algorithm for the weighted matching problem. Information Processing Letters 85, 211–213 (2003)
14. Drake, D.E., Hougardy, S.: Linear time local improvements for weighted matchings in graphs. In: Jansen, K., Margraf, M., Mastrolli, M., Rolim, J.D.P. (eds.) WEA 2003. LNCS, vol. 2647, pp. 107–119. Springer, Heidelberg (2003)
15. Pettie, S., Sanders, P.: A simpler linear time $2/3 - \epsilon$ approximation for maximum weight matching. Information Processing Letters 91, 271–276 (2004)
16. Mehlhorn, K., Näher, S.: LEDA - A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (1999)
17. Soper, A.J., Walshaw, C., Cross, M.: A combined evolutionary search and multilevel optimisation approach to graph-partitioning. J. Global Optimization 29, 225–241 (2004) http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/

Experimental Evaluation of Parametric Max-Flow Algorithms

Maxim Babenko[1,⋆], Jonathan Derryberry[2], Andrew Goldberg[3], Robert Tarjan[2,⋆⋆], and Yunhong Zhou[2]

[1] Moscow State University, Moscow, Russia
[2] HP Labs, 1501 Page Mill Rd, Palo Alto, CA 94304
[3] Microsoft Research – SVC, 1065 La Avenida, Mountain View, CA 94043

Abstract. The *parametric maximum flow problem* is an extension of the classical maximum flow problem in which the capacities of certain arcs are not fixed but are functions of a single parameter. Gallo *et al.* [6] showed that certain versions of the push-relabel algorithm for ordinary maximum flow can be extended to the parametric problem while only increasing the worst-case time bound by a constant factor. Recently Zhang *et al.* [14,13] proposed a novel, simple *balancing algorithm* for the parametric problem on bipartite networks. They claimed good performance for their algorithm on networks arising from a real-world application. We describe the results of an experimental study comparing the performance of the balancing algorithm, the GGT algorithm, and a simplified version of the GGT algorithm, on networks related to those of the application of Zhang *et al.* as well as networks designed to be hard for the balancing algorithm. Our implementation of the balancing algorithm beats both versions of the GGT algorithm on networks related to the application, thus supporting the observations of Zhang *et al.* On the other hand, the GGT algorithm is more robust; it beats the balancing algorithm on some natural networks, and by asymptotically increasing amount on networks designed to be hard for the balancing algorithm.

1 Introduction

The parametric maximum flow problem is a generalization of the ordinary maximum flow problem in which the capacities of arcs out of the source (into the sink) depend on a single parameter and are monotonically increasing (decreasing) functions of the parameter. Applications of parametric maximum flow beyond those of ordinary maximum flow include product selection [3,12], database record segmentation [5], repair kit selection [11], and flow sharing [6].

The current best time bounds for the ordinary maximum flow problem on a network with n vertices, m arcs, and integral arc capacities bounded by U are $O(nm \log_{m/(n \log n)} n)$ [10] and $O(\min\{n^{2/3}, m^{1/2}\} m \log(n^2/m) \log U)$ [7]. The former algorithm is based on the push-relabel method [8]. Gallo *et al.* [6] show

[⋆] Part of this work was done while the author was visiting Microsoft Research – SVC.
[⋆⋆] Also Department of Computer Science, Princeton University.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 256–269, 2007.
© Springer-Verlag Berlin Heidelberg 2007

how to modify certain versions of the push-relabel method using amortization and graph contraction to obtain an algorithm that solves the parametric maximum flow problem yet has the same asymptotic complexity as the original algorithm. Their idea applies to the algorithm of [10], giving an $O(nm \log_{m/(n \log n)} n)$ bound for the parametric flow problem. Tarjan *et al.* [13] give a divide and conquer approach that uses an ordinary maximum flow algorithm as a black box to achieve a running time that is a factor $\min\{n, \log(nU)\}$ worse than that of the black box algorithm. In combination with [7], this gives an $O(\min\{n^{2/3}, m^{1/2}\}m \log(n^2/m) \log U \min\{n, \log(nU)\})$ bound for the parametric problem. In practice, certain implementations of the push-relabel method (*e.g.* [4]) have better overall performance than those of the algorithm of [7], which makes the GGT algorithm a promising choice for the parametric flow problem.

Zhang *et al.* [14] recently introduced an algorithm for the parametric problem based on a new technique called *star balancing* (see Section 3). This algorithm solves the special case of the parametric problem in which the network is bipartite, source arcs have capacity λ, where λ is the parameter, sink arcs have constant capacity, and all other arcs have infinite capacity. This is an important special case, which includes all of the applications mentioned above except for flow sharing. The star balancing algorithm with a small enhancement suggested by Tarjan *et al.* [13] runs in time $O(mn^2 \log(nU))$ [13]. One can show that this analysis is tight for a family of long path examples (See Section 4.2). Although this bound is significantly worse than the best bounds currently known, the worst-case bound is overly pessimistic for many real-world instances. In particular, the star balancing algorithm performs well on several real-world instances of the product selection problem [14]. This motivates experimental comparison between this algorithm and the GGT algorithm.

Few experimental studies of the parametric flow problem have been published in the open literature [2,13,14]. Our codes are the same or better than the corresponding ones in these studies. The only other implementation we are aware of is based on an algorithm described in [9]. However, this implementation became available to us too late for comparison in the current paper.

Our comparison between the GGT and star balancing algorithms involves several steps. First, one needs to develop efficient implementations of the algorithms, which is non-trivial. Then one needs to find interesting real-world and synthetic instances that show strengths and weaknesses of the algorithms. We restricted the experiments described here to bipartite problems of the kind to which the star balancing algorithm applies. Experiments with the GGT algorithm on some other graph types can be found in [2].

The rest of this paper is organized as follows. Section 1.1 reviews the ordinary and parametric maximum flow problems and describes the notations we use. Section 2 describes the GGT algorithm and an efficient implementation of it. Section 3 describes the star balancing algorithm and its implementation. Section 4 is devoted to our experiments. Finally, Section 5 contains concluding remarks, including possible future research directions.

1.1 Background and Notation

For the ordinary maximum flow problem, the input is a directed, capacitated network $\mathcal{N} = (V, A, s, t, c : A \to \mathbb{Z}^+)$, where V is a set of vertices of size n, A is a set of directed arcs (u, v) of size m, s and t are two special vertices (the source and the sink), and c is a capacity function. We assume that the capacities are integers in the range $[1, U]$. A *flow* in a network is a function $f : A \to \mathbb{R}$ that satisfies the capacity constraints $0 \leq f(a) \leq c(a) \quad \forall a \in A$ and the flow conservation constraints $\sum_{(u,v)\in A} f(u, v) = \sum_{(v,w)\in A} f(v, w)$ for all $v \in V - \{s, t\}$. The output for an ordinary maximum flow problem is a flow f such that $\sum_{(s,v)\in A} f(s, v)$ is maximized.

For the parametric maximum flow problem, the input is a directed, capacitated network $\mathcal{N} = (V, A, s, t, c : A \times \mathbb{R} \to \mathbb{Z}^+)$, where the extra input to the capacity function is a parameter λ, upon which the capacities of some arcs may depend. The capacities of arcs out of the source are monotonically increasing in λ, while those of arcs into the sink are monotonically decreasing in λ.[1] All other arcs must have constant capacities (*i.e.*, the capacities cannot depend on λ). The set of minimum cuts for all values of λ has a nested structure: as the value of λ increases, the source side of the cut grows. As a result, there are $n - 1$ or fewer *critical* values of λ, called *breakpoints*, at which the minimum cut changes. The output for a parametric problem is the sequence of breakpoints along with the corresponding nested cuts, and possibly corresponding maximum flows (or information about them).

2 GGT Algorithm

2.1 Push-Relabel Algorithm

The GGT algorithm is based on the push-relabel algorithm [8] for the maximum flow problem. The push-relabel algorithm uses two basic operations, *push* and *relabel*, and maintains a flow and integral *distance labels* on vertices. The important properties of the algorithm are that the distance labels are monotonically increasing, the value of each distance label changes by $O(n)$, and the work of the algorithm is charged to the distance label increases. We assume that the reader is familiar with the push-relabel algorithm as discussed in [8] or [6].

2.2 GGT Algorithm

In this section we describe two algorithms for the parametric flow problem, a simple algorithm based on graph contraction and the GGT algorithm, which also uses amortization to improve the worst-case complexity.

[1] Gallo *et al.* [6] show how to transform a parametric problem so that all of the arcs into the sink are of constant capacity. For simplicity, in the rest of the paper we assume that the arcs into the sink have constant capacity and the arcs out of the source all have capacities that are linear functions of λ.

A simple algorithm for computing all breakpoints works recursively. At each call, the algorithm gets an interval (λ_1, λ_3) and cuts corresponding to λ_1 and λ_3, and outputs all breakpoints in the interval. Initial values of λ_1 and λ_3 that are less than and greater than all breakpoints, respectively, are easy to find (see [6]).

Let $a_1 + \lambda b_1$ and $a_3 + \lambda b_3$ be the parametric capacities of the two input cuts. Set $\lambda_2 = (a_1 - a_3)/(b_3 - b_1)$ and compute the minimum cut corresponding to λ_2. If the parametric capacity of the cut is not equal to $a_1 + \lambda b_1$ or $a_3 + \lambda b_3$, then λ_2 is not a breakpoint, and we recursively find all breakpoints on (λ_1, λ_2) and on (λ_2, λ_3). Otherwise, it is a breakpoint, and we output it. Then, if the capacity is equal to $a_1 + \lambda b_1$, we recurse on the interval (λ_2, λ_3). In the other case, we recurse on (λ_1, λ_2). When making a recursive call for the interval (λ_1, λ_2), we contract the vertices on the sink side of the minimum cut corresponding to λ_2. Similarly, when making the other recursive call, we contract vertices on the source side. Each call of the algorithm is dominated by a minimum cut computation, and one can show that the number of calls is $O(n)$.

Next we describe the GGT algorithm. The algorithm uses amortization. One way to use amortization in the context of the simple algorithm is to note that when recursing on (λ_2, λ_3), one can use the distance labels (on the sink side of the computed cut) from the current flow computation and amortize the cost of such recursive calls over one maximum flow computation. Note that the distance labels on the source side of the cut are "infinite" so the other recursive call cannot be amortized. To obtain the desired bound, the GGT algorithm makes sure that the cost of the flow computation on the bigger graph is amortized.

To achieve this, the algorithm runs two flow computations in parallel; forward from the source and backward from the sink. Assume that the forward computation finishes first; the other case is symmetric. Then if the sink side of the resulting cut has at least as many vertices as the source side, we disregard the result of the backward computation. Otherwise, we finish the backward computation and keep the labels on the source side of the cut, which is at least as big as the sink side. This way the GGT algorithm amortizes the cost of the bigger recursive call at each level, leading to the desired time bound. See [6] for details.

2.3 Implementation Issues

We implemented two versions of the Gallo-Grigoriadis-Tarjan algorithm. The complete version (GGT) uses amortization and bidirectional flow computations. Our implementation uses the gap and global relabeling heuristics (see [4], *e.g.*), but does not use the dynamic tree data structure, so its running time bound is $O(n^2\sqrt{m})$. We also implemented a simple version of the algorithm (SIMP) that starts each maximum flow from scratch and uses the forward computation only. Otherwise the implementation is similar and uses the gap and global relabeling heuristics as well. This implementation has a worse asymptotic bound but smaller constant factors. The efficiency of the resulting implementations requires careful implementation of the contraction operations, including maintaining implicit flows on contracted arcs.

Dealing with precision. The above discussion assumes unlimited precision arithmetic. Because of the multiplicative factors in the parametric cut capacities, one may need high precision to distinguish between adjacent breakpoints. However, using high-precision arithmetic is expensive, and in some applications one may not need to distinguish between breakpoint values that are close together. Our approach is to use 64-bit integer arithmetic and distinguish only between breakpoints that are far enough apart. Our implementation can miss some breakpoints, but for each missed breakpoint we find a value that is close. Note that using (even double precision) floating point arithmetic does not avoid numerical issues and may lead to correctness and termination problems.

Our implementation starts by selecting an integer multiplier M and multiplying all capacities by M. The value of M is selected so that for the highest value of λ the total capacity of arcs from the source is less than 2^{62}, and for the lowest value of λ the same holds for the arcs into the sink. This choice of M guarantees that flow excesses do not exceed 2^{62}, overflow errors will be detected, and our correctness checker, which needs an extra bit of precision, can be implemented.

During the algorithm initialization, when calculating the initial range, we round λ_1 down and λ_3 up to the nearest integer. During the algorithm execution, we round the value of λ_2 down.

Note that because of the rounding, a value x we output may not be a breakpoint. However, the following properties hold. These properties follow from the fact that we evaluate the parametric capacity function at points which are integer multiples of $1/M$.

1. If we output a value x, then there is a breakpoint in the interval $[x-1/M, x+1/M]$.
2. For every breakpoint y, we output a value in $[y-1/M, y+1/M]$.
3. For every two distinct x_1 and x_2 we output, there are corresponding minimum cuts $(X_1, \overline{X_1})$, $(X_2, \overline{X_2})$ such that the parametric capacities of the two cuts are different.

Note that if we restrict the precision of the values we output, then this is the best we can do.

In addition to outputting the approximate breakpoint parameter values, we build a data structure containing the corresponding cuts. Since the cuts are nested, the data structure is an ordered list of vertices, with a pointer to the last vertex of the source-side set for each cut. Note that if all distinct breakpoints are at least $2/M$ apart, the cuts correspond to the true breakpoint values, and can be used to compute the exact breakpoint values.

3 Star Balancing Algorithm

3.1 Algorithm Description

First, we briefly review the star balancing algorithm. For a more detailed description, the reader should consult [14] and [13]. The star balancing algorithm

is an algorithm for solving instances of the parametric maximum flow problem that meet the following constraints:

- the network is *bipartite*, that is, $V \backslash \{s, t\}$ can be partitioned into sets V_1 and V_2 such that all arcs from s are to members of V_1, all arcs to t are from members of V_2, and all other arcs are from members of V_1 to members of V_2
- all arcs from s have capacity λ
- all arcs to t have constant capacity
- all other arcs have infinite capacity.

For each arc (s, u), we define $\lambda(f, u)$ to be the unique value of λ such that $c((s, u), \lambda) = f(s, u)$, and refer to it as u's *λ-value*. Additionally, it will be useful to have notation for describing the changes made to the flow during the process of the algorithm's execution. Define a z-straddling α-move to be the process of starting from an initial flow f and pushing $\alpha > 0$ units of flow along a simple cycle (s, u_1, v, u_2, s) for which $\lambda(f, u_1) + \alpha \leq z \leq \lambda(f, u_2) - \alpha$. Any z-straddling move for any z is defined to be a *balancing move*.

The star balancing algorithm begins by replacing the arcs from the source with arcs of infinite capacity, and then finding an arbitrary maximum flow in the resulting network, which can be done in linear time. Next, the algorithm repeatedly *balances* members v of V_2 by changing the current flow f to a new flow f' via modifying the flows on arcs among $\{(s, u) \cup (u, v) \mid (u, v) \in A\}$ so that there are no remaining balancing moves involving v. Note that if flows are constrained to be integral as is the case in the implementation described in this paper, there may be remaining fractional balancing moves, but no remaining *integral* balancing moves.

Balancing a star can be accomplished in time linear in the degree of the vertex [13]. Balancing can be done in any order, and is repeated until a sufficient stopping condition is reached. Theoretical analysis of the algorithm [13] assumes round-robin balancing (*i.e.*, repeatedly iterating over a list of the members of V_2), although our implementation uses a working set heuristic [14] that is different from simple round-robin balancing (See Section 3.2).

3.2 Implementation Details

Next we describe a few details of our implementation of the star balancing algorithm.

First, although balancing a vertex $v \in V_2$ can be accomplished in time linear in the degree k of v [13] using weighted selection, our implementation uses sorting and takes $O(k \log k)$ time. In practice, we found that using the sorting-based algorithm was just as good as using the linear-time algorithm, probably because only a small amount of time was spent balancing vertices of high degree for the inputs we tried and because the sorting-based algorithm had lower overhead since it uses a library sorting routine.

Second, rather than using round-robin balancing, we used the working set heuristic originally introduced and described in [14]. It does round robin balancing, but if a vertex $v \in V_2$ does not cause the flow to change during an iteration,

it is marked "dead" and left out of future iterations, until all members of V_2 are dead, at which point all members of V_2 are returned to "live" status. The upper bounds proved for round-robin star balancing apply to the working-set variant, and the same pathological long path example shows that this analysis is tight for the working-set variant as well. In practice, the working set heuristic seems to result in a significant speedup on many real-world and synthetic inputs.

Third, the stopping rule that we use for this implementation is slightly different from the stopping rules presented in the theoretical paper [13]. In our implementation, once the working set is empty, all members of V_2 become live, and if one more round of balancing does not change the flow of any arc, then balancing stops. Once balancing stops, we must use the current λ-values to determine the set of breakpoints to report. Two natural options are:

– reporting all distinct λ-values based on the final flow
– reporting the average of λ-values for each *section*, where sections partition the vertices so that every possible remaining balancing move is entirely within a section. (See [13].)

We chose to do the latter, since this guarantees that all reported breakpoints correspond to actual parametric minimum cuts.

3.3 Precision Issues

For a practical implementation, precision issues are important. One option is to work with high-precision or rational numbers. Both of these options introduce significant overhead compared to the use of hardware arithmetic operations, so we instead opted to use 64-bit integers as used in the other two implementations of this paper. In what follows, we discuss the most important issues that arise when using limited precision in the star balancing algorithm.

First, we address the question of how much precision is needed to solve the problem exactly, assuming the fixed capacities are integral. It can be shown that for two distinct breakpoints, λ' and λ'', it is the case that $|\lambda' - \lambda''| \geq 4/|V_1|^2$. This implies that multiplying arc capacities by some multiplier $M > |V_1|^2/2$ ensures that all true breakpoints differ by more than 2, so that if no augmenting path $(u_1, v_1, u_2, v_2, \ldots, v_{k-1}, u_k)$ remains (where $u_i \in V_1$ and $v_i \in V_2$) along which 1 unit of flow can be pushed so as to decrease $|\lambda(f, u_1) - \lambda(f, u_k)|$, then all reported breakpoints will be true breakpoints, and vice versa.

Because balancing only guarantees the non-existence of balancing *moves* (*i.e.*, augmenting paths of two arcs, excluding the two arcs from s), however, and does *not* guarantee that there are no remaining balancing *paths* as described above, such a multiplier is insufficient to guarantee that exactly the true breakpoints are found by the star balancing algorithm. Because of this weakness, pathological examples show that true breakpoints can be up to a distance $\Omega(|V_1|)$ from the reported breakpoints if we only report one λ-value per section.[2] Hence, an

[2] It should be noted that we can eliminate all balancing paths and achieve a precision guarantee identical to those of the SIMP and GGT implementations if we add a

additional multiplicative factor of $|V_1|$ is required (and sufficient, since no section at the end of the star balancing phase can have λ-values that differ by more than $|V_1| - 1$) to guarantee that exactly the true breakpoints are reported.

Whatever multiplicative factor M is used for the capacities, the star balancing algorithm may work with λ-values as high as $M \cdot |V_2| \cdot U$, so this value must fit into a 64-bit integer, which we use to store λ-values. In these experiments, we used a multiplier of $|{V_1}^3|$.

If we are not concerned with finding exactly the true breakpoints, we can use a smaller multiplier and report all λ-values (scaled down by M) as breakpoints at the termination of the balancing phase. Using this approach, there will be some reported breakpoint within $1/(2M)$ of each true breakpoint, but a pathological example can be constructed that shows that reported breakpoints can be as far as $\Omega(\log n/(M \log\log n))$ from the closest true breakpoint. We believe that this lower bound on inaccuracy is tight.

4 Experimental Comparison

In this section we report on experimental performance of SIMP, GGT, and the star balancing algorithm, called SB. These implementations use the same language (C++), compiler (cygwin), optimization flags (-O4), and were run on the same computer, a Hewlett-Packard desktop with a 3.2GHz Pentium 4 processor and 2GB of RAM. However, SIMP and GGT were implemented by a different set of people than those that implemented SB. Also, while these algorithms are for the general problem, SB works only for the special case discussed earlier.

The inputs we used in this experiment were a combination of real and synthetic data. The real data we used were the inputs used in [14]. These datasets are instances of the revenue optimization problem, and correspond to sets of products and orders for various subsets of these products.

In addition, we created synthetic datasets corresponding to each of the real datasets. We computed the degree distributions of vertices on the left and right sides of the bipartitions, and the distribution of the capacities of arcs going into the sink, and used these to generate synthetic networks with statistics similar to each real dataset. The purpose of these synthetic datasets was to examine whether any underlying structure of the problems may have been affecting the relative running times of the algorithms.

We created various other simple synthetic examples to illuminate the degree to which the pathological worst-case running time of SB occurs on various simple problem instances related to the long path example as compared to the more

Simple post-processing step that repeatedly pushes one unit of flow along augmenting paths from $u_1 \in V_1$ to $u_2 \in V_1$ so as to balance their λ-values. This post-processing step runs in worst-case time $O(m|V_2|^2)$ but may run significantly faster in practice. Although the post-processing step does not worsen the worst-case running time of the star balancing algorithm, we chose not to incorporate it into the implementation and instead leave the less-than-ideal precision guarantee as part of the specification of the implementation.

robust SIMP and GGT implementations. We started with various lengths of the long path example with uniform sink arc capacities, and generalized this type of example to have variable sink arc capacities and extra arcs, both random and nonrandom, between the left and right sides of the bipartition. We also experimented with some natural problem variants.

4.1 Real Data and Its Synthetic Model

First we compare the implementations on real-life problem instances from [14]. There are four datasets, taken from the same real-world application. See Table 3 for the vertex and arc sizes of these datasets. The results of the experiments, displayed in Fig. 1 and Table 1, show that SB, despite its inferior worst-case running time, outperforms GGT and SIMP for these datasets.

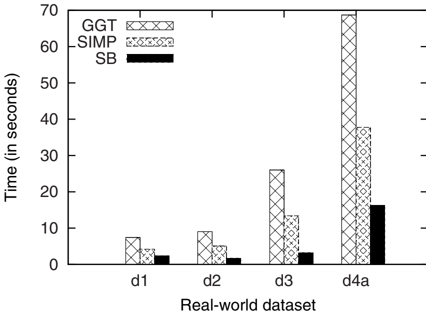


Fig. 1. A comparison of the running times of the GGT, SIMP, and SB implementations on real-world inputs

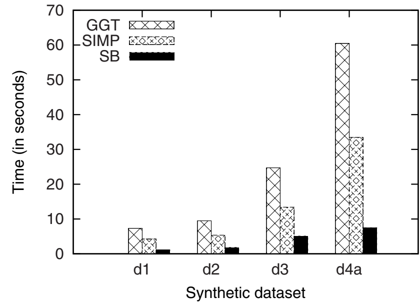


Fig. 2. Comparison results on synthetic datasets of the same sizes, degree distributions, and capacity distributions as the corresponding real-world inputs

Table 1. Tabular data corresponding to Fig. 1

	d1	d2	d3	d4a
GGT	7.41	9.04	25.98	68.68
SIMP	4.21	5.07	13.41	37.75
SB	2.41	1.74	3.27	16.29

Table 2. Tabular data corresponding to Fig. 2

	d1	d2	d3	d4a
GGT	7.32	9.48	24.71	60.44
SIMP	4.22	5.37	13.41	33.45
SB	1.12	1.83	5.11	7.49

To help understand why this is happening, we implemented a synthetic problem generator that models these real-life problems, as discussed earlier. This provided some robustness to the results that used the real data. The results of these experiments, given in Fig. 2 and Table 2, show that the performance gap remains roughly the same.

Table 3. A description of the problem sizes of the real-world problem instances of the product selection problem used in these experiments

	d1	d2	d3	d4a
$\|A\|$	454k	625k	1,401k	3,386k
$\|V_1\|$	263	232	344	439
$\|V_2\|$	39k	53k	123k	286k

Table 4. Tabular data corresponding to Fig. 5

	6400	12800	25600	51200	102400	204800	409600
GGT	0.51	1.22	2.81	6.35	14.41	32.11	70.58
SIMP	0.23	0.57	1.35	3.07	7.04	15.76	34.44
SB	0.48	1.14	2.66	5.88	13.21	30.91	77.86

4.2 The Long Path Example and Its Variations

One type of bad example for SB is one in which there is a long path over which many iterations of balancing are required to propagate modest changes in λ-values through the graph. Tarjan *et al.* [13] showed that the running time of the balancing algorithm using round-robin balancing on examples such as the one shown in Fig. 3 is $\Omega(n^3 \log n)$. This is troubling, especially considering that such examples are extremely easy for most parametric maximum flow algorithms to solve. Indeed, as the experimental results show in Fig. 4, SB performs drastically worse on this family of examples than GGT and SIMP.

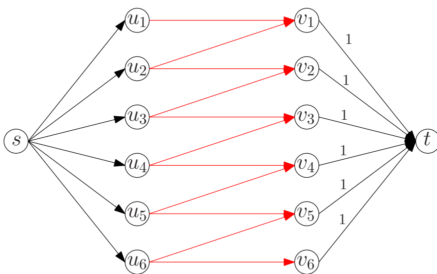


Fig. 3. The long path example, which with unlucky initialization requires $\Omega(n^3 \log n)$ time to finish balancing

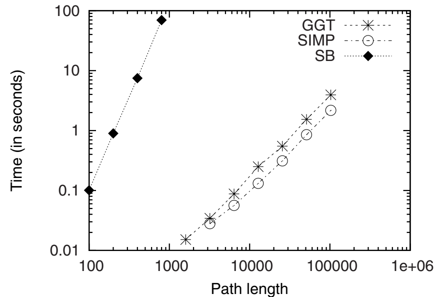


Fig. 4. A comparison of the running times of GGT, SIMP, and SB on increasingly large long path examples. Note that the running time of GGT and SIMP grows roughly linearly, while the running time of SB grows roughly cubically.

The behavior of the balancing algorithm on inputs resembling long paths is troubling, but why might long paths not be a problem in practice? One intuitive explanation for this is that real data may have variability in the capacities of the arcs incident to the sink. Intuitively, this variability can keep SB from needing to push flow over long distances to reach the balanced state. Using capacities distributed uniformly at random on [1, 1000], this intuition was confirmed as

shown in Fig. 5 and Table 4, where such variability improves the performance of SB almost to that of GGT. Fig. 6 and Table 5 also show how the competitiveness of SB increases as the variability of the sink arc capacities increases.

Another reason why long paths may not be a problem in practice is that additional connections in the graph may ameliorate the problem. For example, if a random matching is overlaid on top of a long path graph (*e.g.* Fig. 3), the long path remains but there are many shortcuts for flow to take to reach one end of the long path from the other end. The results, shown in Fig. 7 and Table 6, indicate that this variation on the long path example also eliminates much of the difference in performance between SB and the other two implementations.

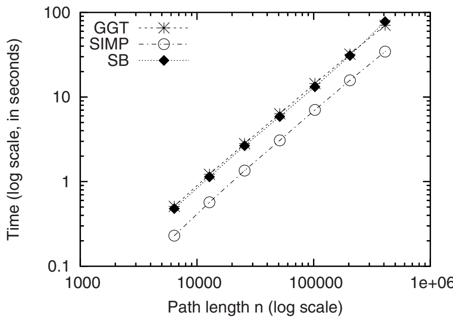


Fig. 5. A comparison of the running times of the GGT, SIMP, and SB implementations on long path inputs (as in Fig. 3) with sink capacities drawn from [1, 1000] uniformly at random. See Table 4 for the data corresponding to this graph.

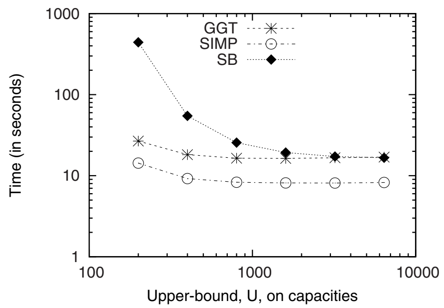


Fig. 6. A comparison among the algorithms on a long path example with 102,400 vertices and capacities distributed uniformly in $[100, U]$. See Table 5 for the data corresponding to this graph.

In fact, as Fig. 8 and Table 7 show, adding additional random matchings continues to improve the performance of SB algorithm relative to those of GGT and SIMP. This leads one to speculate that it might be possible to prove a running time bound that depends on some kind of expansion property of the underlying bipartite graph. This is discussed in Section 5.

Another way to view the long path example is as a 1-dimensional checkerboard in which red squares correspond to members of V_1 and black squares correspond

Table 5. Tabular data corresponding to Fig. 6

	200	400	800	1600	3200	6400
GGT	26.77	18.19	16.44	16.35	16.65	16.98
SIMP	14.30	9.21	8.29	8.13	8.11	8.25
SB	442.10	54.64	25.56	19.31	17.23	16.63

Table 6. Tabular data corresponding to Fig. 7

	6400	12800	25600	51200	102400	204800	409600
GGT	0.14	0.31	0.67	1.37	2.81	5.78	12.08
SIMP	0.08	0.19	0.40	0.84	1.72	3.58	7.42
SB	0.21	0.71	1.24	3.10	6.93	24.38	111.44

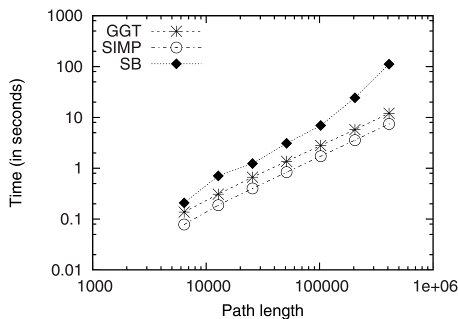


Fig. 7. A comparison of the running times among GGT, SIMP, and SB on long path inputs (as in Fig. 3) in which a random matching is overlaid on top of the long path

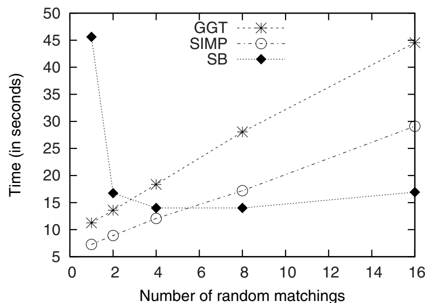


Fig. 8. A comparison of the running times among GGT, SIMP, and SB on long path inputs in which k random matchings are overlaid on top of the long path. The number of vertices in the path was fixed to 409,600.

to members of V_2 and there are arcs from each member of V_1 to the members of V_2 corresponding to adjacent squares on the checkerboard. Based on this view, we can extend the long path example to higher-dimensional checkerboards in which the paths are not as long for graphs of roughly the same size (See Fig. 9). More specifically, in a d-dimensional checkerboard example with n vertices, the diameter of the graph is $\Theta(n^{1/d})$.

Indeed, as we increase the number of dimensions of the checkerboard while holding the number of vertices roughly constant, the performance of SB improves relative to that of the GGT and SIMP (See Fig. 10 and Table 8).

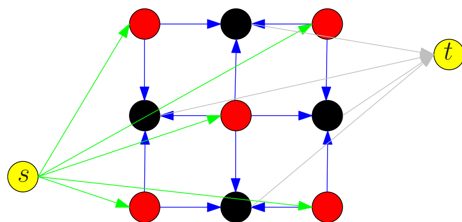


Fig. 9. An example of a 3 × 3 two-dimensional checkerboard example

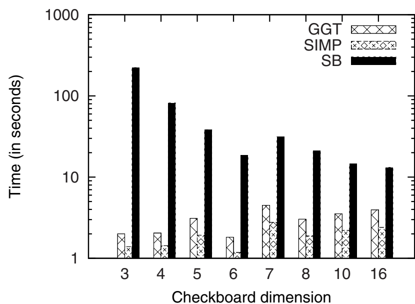


Fig. 10. A comparison of the running times of the GGT, SIMP, and SB implementations on hypercube checkerboard inputs of similar size (about 60,000 vertices each) but increasing dimension

Table 7. Tabular data corresponding to Fig. 8

	1	2	4	8	16
GGT	11.24	13.59	18.35	28.05	44.53
SIMP	7.26	8.91	12.07	17.19	29.12
SB	45.62	16.75	14.00	14.00	16.93

Table 8. Tabular data corresponding to Fig. 10

	3	4	5	6	7	8	10	16
GGT	2.00	2.05	3.10	1.82	4.49	3.04	3.53	3.95
SIMP	1.39	1.43	1.92	1.18	2.77	1.87	2.20	2.40
SB	223.05	82.00	38.39	18.56	31.29	21.08	14.59	13.18

5 Conclusions and Future Work

Our comparison of the push-relabel algorithms SIMP and GGT with the star balancing algorithm shows that no algorithm dominates the others. This is despite the fact that the push-relabel algorithms have significantly better worst-case bounds. Also, SIMP outperforms GGT in all our experiments. The push-relabel codes are more robust – when they are slower, they are not slower by as much – probably due to the better worst-case bound. For real-life instances from the one application domain we tried, the balancing algorithm was fastest, confirming the earlier claims of Zhang *et al.* [14,13] based on indirect estimates of GGT performance. In addition, the star balancing algorithm is easier to implement.

Our results show that the pathological behavior of the balancing algorithm when running on long path examples disappears as various changes are made to the network , such as adding sink arc capacity variability, adding random edges, or parameterizing the dimension of the long path example so as to extend it to a higher number of dimensions. This suggests two directions to take for future work regarding using the balancing framework for parametric max-flow.

First, it is clear that the pathological long path behavior is moderated when additional connections between the two sides of the partition provide shortcuts to the long path, or when the long path is cut by variable sink arc capacities. This suggests proving a bound better than the existing bound when the graph has sufficient expansion or some other property. Proving such a bound would be nice in that we would be able to give a better guarantee on the running time of this extremely simple algorithm. Second, in addition to proving a better bound, it would be interesting to see if there were a way to remove the pathological long path behavior by devising a hybrid algorithm that is fast on inputs with long paths, and remains fast on the types of inputs for which the balancing algorithm shows good performance. Such an algorithm would probably only be interesting if it did something other than running two different algorithms in parallel, and it may even be possible to prove a better worst-case running time for such a hybrid algorithm than the current best known worst-case running time.

For the push-relabel algorithms, it seems hard to construct a worst-case example. In fact, it is hard to construct an example on which GGT is significantly faster than SIMP, which should be the case in the worst-case example if the sophisticated amortization used by GGT is needed to achieve the time bound. The only known example where GGT beats SIMP [2] uses the fact that the underlying push-relabel algorithm is asymmetric and can take very different time when solving the equivalent problem on the reverse graph. The GGT algorithm

runs on the original and the reverse graph in parallel and on such an example it is faster for this reason. An interesting question is whether the clever amortizations used in GGT is reflected in practice. For example, it would be interesting to see if there exist instances of the parametric maximum flow problem for which GGT and SIMP have similar running times if the parameter value is fixed to any value, but GGT saves a logarithmic factor over SIMP when the entire parametric problem is solved all at once. Finally, it would be interesting to see if there is a faster implementation of a push-relabel algorithm for the special bipartite version of the problem studied in this paper. In this regard, see [1,6] for results on bipartite maximum flow.

References

1. Ahuja, R.K., Orlin, J.B., Stein, C., Tarjan, R.E.: Improved algorithms for bipartite network flow. SIAM Journal on Computing 23(5), 906–933 (1994)
2. Babenko, M.A., Goldberg, A.V.: Experimental evaluation of a parametric flow algorithm. Technical report, Microsoft Research (2006)
3. Balinski, M.L.: On a selection problem. Management Science 17(3), 230–231 (1970)
4. Cherkassky, B.V., Goldberg, A.V.: On Implementing Push-Relabel Method for the Maximum Flow Problem. Algorithmica 19, 390–410 (1997)
5. Eisner, M.J., Severance, D.G.: Mathematical techniques for efficient record segmentation in large shared databases. J. ACM 23(4), 619–635 (1976)
6. Gallo, G., Grigoriadis, M.D., Tarjan, R.E.: A fast parametric maximum flow algorithm and applications. SIAM J. Comput. 18(1), 30–55 (1989)
7. Goldberg, A.V., Rao, S.: Beyond the flow decomposition barrier. J. ACM 45(5), 783–797 (1998)
8. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM 35(4), 921–940 (1988)
9. Hochbaum, D.S.: The Pseudoflow Algorithm and the Pseudoflow-Based Simplex for the Maximum Flow Problem. In: Bixby, R.E., Boyd, E.A., Ríos-Mercado, R.Z. (eds.) Integer Programming and Combinatorial Optimization. LNCS, vol. 1412, pp. 325–337. Springer, Heidelberg (1998)
10. King, V., Rao, S., Tarjan, R.: A Faster Deterministic Maximum Flow Algorithm. J. Algorithms 17, 447–474 (1994)
11. Mamer, J., Smith, S.: Optimizing field repair kits based on job completion rate. Management Science 28(11), 1328–1333 (1982)
12. Rhys, J.M.W.: A selection problem of shared fixed costs and network flows. Management Science 17(3), 200–207 (1970)
13. Tarjan, R., Ward, J., Zhang, B., Zhou, Y., Mao, J.: Balancing applied to maximum network flow problems. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 612–623. Springer, Heidelberg (2006)
14. Zhang, B., Ward, J., Feng, Q.: Simultaneous parametric maximum flow algorithm with vertex balancing. Technical Report HPL-2005-121, HP Labs (2005)

Experimental Study of Geometric t-Spanners: A Running Time Comparison

Mohammad Farshi[1,*] and Joachim Gudmundsson[2]

[1] Department of Mathematics and Computing Science, TU Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
m.farshi@tue.nl
[2] NICTA[**], Sydney, Australia
joachim.gudmundsson@nicta.com.au

Abstract. The construction of t-spanners of a given point set has received a lot of attention, especially from a theoretical perspective. We experimentally study the performance of the most common construction algorithms for points in the Euclidean plane. In a previous paper [10] we considered the properties of the produced graphs from five common algorithms. We consider several additional algorithms and focus on the running times. This is the first time an extensive comparison has been made between the running times of construction algorithms of t-spanners.

1 Introduction

Consider a set V of n points in the plane. A network on V can be modeled as an undirected graph G with vertex set V of size n and an edge set E of size m where every edge $e = (u, v)$ has a weight $wt(e)$. A geometric (Euclidean) network is a network where the weight of the edge $e = (u, v)$ is the Euclidean distance $|uv|$ between its endpoints u and v. Let $t > 1$ be a real number. We say that a geometric network $G(V, E)$ is a (*geometric*) *t-spanner* for V, if for each pair of points $u, v \in V$, there exists a path in G between u and v of weight at most $t \cdot |uv|$. We call this path a t-path between u and v. The minimum t such that G is a t-spanner for V is called the stretch factor, or dilation, of G. Finally, a subgraph G' of a given graph G is a t-spanner for G if for each pair of points $u, v \in V$, there exists a path in G' of weight at most t times the weight of the shortest path between u and v in G.

Complete graphs represent ideal communication networks, but they are expensive to build; sparse spanners are low-cost alternatives. The weight of the spanner is a measure of its sparseness; other sparseness measures include the number of edges, the maximum degree, and the number of crossings. Spanners for complete Euclidean graphs as well as for arbitrary weighted graphs find applications in robotics, network topology design, distributed systems, design of parallel machines, and many other areas and have been a subject of

* Supported by Ministry of Science, Research and Technology of I. R. Iran.
** National ICT Australia Ltd. is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 270–284, 2007.
© Springer-Verlag Berlin Heidelberg 2007

considerable research. Recently low-weight spanners found interesting practical applications in areas such as metric space searching [16] and broadcasting in communication networks [14]. Well-known theoretical results also use the construction of t-spanners as a building block, for example, Rao and Smith [17] made a breakthrough by showing an optimal $\mathcal{O}(n \log n)$-time approximation scheme for the Euclidean *traveling salesperson problem*, using t-spanners (or banyans). The problem of constructing spanners has received considerable attention from a theoretical perspective, see the recent book by Narasimhan and Smid [15], but almost no attention from a practical or experimental perspective [10, 16, 18].

In this paper we consider the most well-known algorithms for the construction of t-spanners in the plane: variants of greedy spanners and Θ-graphs, spanners constructed from the well-separated pair decomposition (WSPD), skip-list spanners, sink spanners and some hybrid algorithms. Due to the space limitation we only compare the running times of these algorithms (in the full version the graph properties are also studied) for point sets of size up to 10K points, four different distributions; only two are discussed in this paper, and with values of t between 1.1 and 2. The properties of the standard greedy graph, the (ordered) Θ-graph, the WSPD-graph and the hybrid graphs were discussed in [10].

The paper is organized as follows. Next we briefly go through the desirable properties for t-spanners. In Section 2 we describe the implemented algorithms together with the theoretical bounds and implementation details. In Section 3 we discuss the results and finally discuss possible improvements and future research.

Throughout the paper t will be assumed to be a small constant. In the experiments we used values of t between 1.1 and 2. For larger values of t one can use the Delaunay triangulation which is known to have dilation ≈ 2.42 [13].

1.1 Spanner Properties

As input we are given a set V of n points in the plane and a real value $t > 1$. The aim is to compute a t-spanner for V with some good properties where the quality measurements that one consider are as follows:

Size: The number of edges in the graph. This is the most important measurement and all the implemented algorithms produce spanners with $\mathcal{O}(n)$ edges.

Degree: The maximum number of edges incident to a vertex.

Weight: The weight of a Euclidean network G is the sum of the edge weights. The best that can be achieved is a constant times the weight of the minimum spanning tree, denoted $wt(MST(V))$.

Spanner Diameter: Defined as the smallest integer d such that for any pair of vertices u and v in V, there is a path of length at most $t \cdot |uv|$ between u and v containing at most d edges.

2 Spanner Construction Algorithms

Here we give a short description of each of the implemented algorithms together with their theoretical bounds. Note that some of the properties are competing,

e.g., a graph with constant degree cannot have constant spanner diameter, and a graph with small spanner diameter cannot have a linear number of edges [2].

2.1 The Original Greedy Algorithm and an Improvement

The greedy algorithm was discovered independently by Bern in 1989 and Althöfer et al. [1]. The graph constructed using the greedy algorithm will be called a greedy graph. The original algorithm starts with the complete graph G while maintaining a partial spanner graph G' of G. All the edges of G are sorted with respect to their length in increasing order. Next the edges are processed in sorted order. Processing an edge (p, q) entails a shortest path query in G' between p and q. If there is no t-path between p and q in G' then (p, q) is added to G' otherwise it is discarded. The time complexity of the original greedy algorithm is $\mathcal{O}(n^3 \log n)$ and it uses $\mathcal{O}(n^2)$ space.

In [10] we proposed a modifications of the greedy algorithm, denoted improved greedy, that we conjectured should have a running time of $\mathcal{O}(n^2 \log n)$. The idea is that every time a shortest path query is performed from p to q Dijkstra's algorithm computes the shortest distance from p to all other points in V. Instead of neglecting all this information we store it in a matrix. When the next shortest path query, say between u and v, is performed we first look in the matrix if there is a t-path between u and v; if there is then discard (u, v) otherwise perform the query on G' as above. We experimentally compare the running time of the original greedy algorithm with the modified version.

Implementation. The implementations of the algorithms are straight-forward. The shortest path queries are done by using the `Dijkstra` function in LEDA.

2.2 The Approximate Greedy Algorithm

In [10] only the original greedy implementation was considered. It was shown that the quality of the networks produced by the greedy algorithm was superior to the other approaches in terms of number of edges, weight and degree. However, a naïve implementation of it has a running time of $\mathcal{O}(n^3 \log n)$, thus any approach that can speed-up the algorithm would be of great interest. The running time is mainly due to the fact that $\Theta(n^2)$ shortest path queries needed to be answered in a graph with $\mathcal{O}(n)$ edges, each of which could take $\mathcal{O}(n \log n)$ time.

Das and Narasimhan [8] showed how to use clustering to speed up shortest path queries. The approximate greedy algorithm starts with a $\sqrt{t/t'}$-spanner G' with $\mathcal{O}(n)$ edges and constant degree generated by an $\mathcal{O}(n \log n)$-time algorithm. Note that this network does not have to have small weight. Then it computes a $\sqrt{tt'}$-spanner of G' using an approximate variant of the greedy algorithm. To obtain $G(V, E)$ from G' the approximate algorithm starts with $E = \emptyset$ and adds all the short edges (i.e. those of length at most D/n, where D is the distance between the farthest pair of points) to E. For the remaining edges, the algorithm sorts them by increasing weight and processes them in $\log n$ phases. Processing an edge $e = (u, v)$ entails a shortest path query which is answered by performing an

approximate shortest path query on a "cluster graph" H, which is simultaneously maintained. The cluster graph H has the following properties:

1. distances in H "closely" approximate distances in the current graph G'.
2. every vertex in H has bounded degree, and
3. "specialized" shortest path queries in H can be answered in constant time.

For more details see [8] or [15]. The time complexity of this algorithm is $\mathcal{O}(n \log^2 n)$. Note that the graph generated by this algorithm is an approximate version of the graph generated by the original greedy algorithm since the algorithm prunes a graph with linear number of edges and answers shortest path queries using an approximate shortest path query procedure.

Gudmundsson et al. [11] later improved the running time to $\mathcal{O}(n \log n)$ but the modified version is quite involved and therefore we decided to only implement the above version. The following theorem states the theoretical bounds.

Theorem 1. *The approximate greedy graph is a t-spanner of V with $\mathcal{O}(n/(t-1)^3)$ edges, $\mathcal{O}(\frac{1}{(t-1)^3})$ maximum degree and weight $\mathcal{O}(wt(MST(V))/(t-1)^4)$, and can be computed in time $\mathcal{O}(\frac{n}{(t-1)^7} \log n)$.*

Implementation. The initial $\sqrt{t/t'}$-spanner G' was constructed using the sink-spanner algorithm (Section 2.7). This guarantees that the number of edges is $\mathcal{O}(n)$ and that the graph has constant degree. We implemented a variant of Dijkstra's algorithm which answers shortest path queries in constant time in the cluster graph. The query time can be achieved since the maximum degree of the cluster graph is constant and there is a constant upper bound B on the number of edges along a shortest path in the cluster graph, thus we may discard any path containing more than B edges in the priority queue. The bound B can be obtained by choosing the size of the clusters in the cluster graph appropriately.

2.3 The Θ-Graph

The Θ-graph was discovered independently by Clarkson [7] and Keil [12]. Keil only considered the graph in two dimensions while Clarkson extended his construction to also include three dimensions.

Initially we set θ such that $t = \frac{1}{\cos\theta - \sin\theta}$. For each point $u \in V$ consider k non-overlapping cones, $C_i, 1 \leq i \leq k$, with angle $\theta = \frac{2\pi}{k}$ and with apex u. For each cone C_i we add an edge between u and the point within C_i whose orthogonal projection onto the bisector of C_i is closest to u. Note that instead of the bisector of C_i, we can use any line in the cone passing through the apex of the cone. We use one of the boundary lines of the cone instead of the bisector.

Theorem 2. *The Θ-graph is a t-spanner of V for $t = \frac{1}{\cos\theta - \sin\theta}$ with $\mathcal{O}(kn)$ edges and can be computed in $\mathcal{O}(kn \log n)$ time.*

Implementation. To implement the Θ-graph algorithm, we need a dynamic data structure, see [15] for more details, that can perform a point query in a cone in $\mathcal{O}(\log n)$ time. This data structure is implemented using red-black trees. Since

there is no dependence between the cones, one can work on one cone direction at a time, which means that in practice only $\mathcal{O}(n)$ work space is needed.

A problem that we do not consider in the Θ-graph implementation is rounding errors, which may cause some edges not to be added. For example, if a point lies on the boundary of an, otherwise empty, cone then a small rounding error may "move" the point outside the cone. One way to get rid of this error is to use exact arithmetics. A different possibility is to allow the cones to slightly overlap.

2.4 The Ordered Θ-Graph

A simple variant of the Θ-graph that has been shown to have good theoretical performance is the *ordered Θ-graph* by Bose et al. [5]. An ordered Θ-graph of V is obtained by inserting the points of V in some order. When a point p is inserted, we draw the cones around p and connect p to the previously inserted point with closest orthogonal projection in each cone, like the Θ-graph algorithm.

The order is decided as follows. Initially choose an arbitrary vertex $v_n \in V$ and set its order to n, i.e. this is the last point that will be added to the graph. Process v_n by placing k cones with apex at v_n and then adding the edges as in the Θ-graph algorithm. In a generic step, assume we have processed $i-1$ vertices. In the ith step, choose a point with maximum degree from $V - \{v_n, \ldots, v_{n-(i-1)}\}$ and set its order to $n-i$ and then process v_{n-i} assuming that we have the point set $V - \{v_n, \ldots, v_{n-i+1}\}$. This decides an order on the point set.

Theorem 3. *The ordered Θ-graphs is a t-spanner of V for $t = \frac{1}{\cos\theta - \sin\theta}$ with $\mathcal{O}(kn)$ edges and $\mathcal{O}(k \log n)$ degree, and can be computed in $\mathcal{O}(kn \log n)$ time.*

Implementation. For the implementation we use a data structure which is somewhat more complicated than the data structure used for the Θ-graph, since we require the structure to allow for deletions. Due to [5], we use k range trees, one for each cone with apex at the origin. In each range tree we store all points represented in the coordinate system of the two boundaries of the cone. To find the suitable point in a cone with apex at u, it is sufficient to perform a range query with coordinates of u as keys and choose the suitable point between the points reported by the query. We add one extra pointer to each node of the range tree which shows the point with minimum y (or x) coordinate in the subtree. Using this pointer, we can find the suitable point without going through all reported points of the range query. Each range query requires $\mathcal{O}(\log^2 n)$ time, so the total time complexity of the implemented algorithm is $\mathcal{O}(n \log^2 n)$ which is slightly more than the theoretical time bound but much simpler to implement.

In each step of the ordered Θ-graph algorithm the node with maximum degree has to be selected. To find this point, we used a priority queue of all the points. Initially all the nodes have priority n. When an edge (p, q) is added to the partial spanner graph, the priority of p and q is decreased by 1. The point with minimum priority in the queue is the point with maximum degree in the graph.

There is a major difference between the Θ-graph algorithm and the ordered Θ-graph algorithm when it comes to the space complexity. One can construct

the Θ-graph by working on one cone direction at a time, while the ordered Θ-graph algorithm requires us to keep all the cones (range trees) in memory. This is due to the fact that the order is not known in advance. During the processing of one node, we need to check all the cones and add edges if necessary, thus $\Theta(kn)$ space is needed. For small values of t this might cause a major problem. To be more precise, the Θ-graph algorithm used roughly 2% of the memory when constructing a 1.05-spanner on a set with 10,000 points, while the ordered Θ-graph algorithm used almost 85%.

2.5 The Random Ordered Θ-Graph

The ordered Θ-graph algorithm inserts points into the graph in a specific order. However, if the points are processed in random order then the spanner diameter will be bounded by $\mathcal{O}(\log n)$ with high probability [5]. Unfortunately, the degree bound does not hold in this case. There are two reasons why we implemented the random Θ-graph. (1) Random ordered Θ-graphs and skip-list spanners (Section 2.8) are the only two spanners guaranteed to have bounded spanner diameter. Thus a comparison in practice between the two graphs is interesting. (2) Since the vertices are processed in random order we may fix a random order at the beginning which implies that the algorithm only requires $\mathcal{O}(n)$ space, compared to $\mathcal{O}(kn)$ space needed to construct ordered Θ-graphs.

Implementation. The implementation is the same as for the ordered Θ-graph. We only make a random permutation on the input point set and then process the points in the order they appear in after permutation.

2.6 The WSPD-Graph

The well-separated pair decomposition (WSPD) was developed by Callahan and Kosaraju [6].

Definition 1. *Let $s > 0$ be a real number and let A and B be two finite sets of points in $\mathbb{R}^d$. We say that A and B are* well-separated *with respect to s, if there are two disjoint d-dimensional balls C_A and C_B, having the same radius, such that (i) C_A contains A, (ii) C_B contains B, and (iii) the distance between C_A and C_B is at least s times the radius of C_A.*

Definition 2. *Let V be a set of n points in $\mathbb{R}^d$, and let $s > 0$ be a real number. A WSPD for V with respect to s is a sequence of pairs of non-empty subsets of V, $\{A_i, B_i\}_{i=1}^m$, such that (i) A_i and B_i are well-separated w.r.t. s, for all $i = 1, \ldots, m$. (ii) for any two distinct points p and q of V, there is exactly one pair $\{A_i, B_i\}$ in the sequence, such that $p \in A_i$ and $q \in B_i$, or $q \in A_i$ and $p \in B_i$. The integer m is called the* size *of the WSPD.*

Callahan and Kosaraju showed that a set of well-separated pairs of size $m = \mathcal{O}(s^d n)$ can be computed in $\mathcal{O}(s^d n + n \log n)$ time. Constructing a t-spanner using the WSPD is surprisingly easy. It is sufficient to compute a WSPD of V w.r.t. $s = \frac{4(t+1)}{t-1}$ and then for every well-separated pair (A, B) in the WSPD an edge is added between an arbitrary point in A and an arbitrary point in B.

Theorem 4. *The WSPD-graph is a t-spanner for $V \subset \mathbb{R}^2$ with $\mathcal{O}((\frac{t}{t-1})^2 n)$ edges, $\mathcal{O}(\log n \cdot wt(MST(V)))$ weight and can be constructed in time $\mathcal{O}((\frac{t}{t-1})^2 n + n \log n)$.*

We also implemented two versions of the algorithm to improve the degree [2] and the spanner diameter [3]. However, since the experiments showed no improvements for any of these properties we decided not to include them in this paper.

Implementation. We used a split tree for the construction of the WSPD. The points stored at a node is partitioned into two sets by partitioning the non-empty bounding box along its longest side into two boxes of equal size. The tree construction only requires a few percent of the total running time in all our tests.

To decide in constant time if two sets are well-separated we save the smallest enclosing circle of the points in each node. However, their smallest enclosing circles may have different radius so one way to handle this is to say that two sets are well-separated w.r.t. s if the distance between the smallest enclosing circles is at least s times the maximum radius of the two smallest enclosing circles.

2.7 The Sink-Spanner

The sink-spanner construction was defined by Arya et al. in [2] which construct t-spanners with constant degree. The main idea is as follows. We start with a directed $\sqrt{t}$-spanner with bounded out-degree, denoted $\overrightarrow{G}$. We will use the Θ-graph which easily can be seen to have out-degree k, but linear in-degree. For each vertex q in $\overrightarrow{G}$, replace every "star" (the subgraph consisting of all edges in $\overrightarrow{G}$ pointing to q) in $\overrightarrow{G}$ by a $\sqrt{t}$-q-sink spanner. A $\sqrt{t}$-q-sink spanner is a directed graph where each point has a directed $\sqrt{t}$-path to q. It can be obtained by processing each node q in $\overrightarrow{G}$ as follows. Consider all points which have an edge pointing to q. Let A_q be the set of all such a nodes. We replace all the edges pointing to q by a $\sqrt{t}$-path using the partial sink spanner procedure. In the partial sink spanner procedure we look at k cones with apex at q and we partition the points in A_q based on the cones. Let S_i be the points in the ith cone. For each cone i, add an edge between q and the closest point in S_i, say q_i, and then recurse on the partial sink spanner procedure on q_i and $S_i \setminus \{q_i\}$. In the case that one cone contains more than half of the points, split the points in the cone to two almost equal parts and do the same thing as above. This guarantees that the subproblems half in size, thus we get:

Theorem 5. *The sink-spanner is a t-spanner for $V \subset \mathbb{R}^2$ with $\mathcal{O}(kn)$ edges and $\mathcal{O}(\frac{1}{(t-1)^2})$ maximum degree, and can be constructed in time $\mathcal{O}(kn \log n)$.*

Implementation. To construct the first directed $\sqrt{t}$-spanner, we use the Θ-graph algorithm, with the modification that we add directed edges instead of undirected edges. Then for each node q in the directed graph, we look at all points which has an edge pointing to q. Let A_q be the set of all such a nodes. We replace all the edges pointing to q by a $\sqrt{t}$-path using the partial sink spanner

procedure. In the partial sink spanner procedure we look at k cones with apex at q and we partition the points in A_q based on the cones. Let S_i be the points in the ith cone. For each cone i, add an edge between q and the closest point in S_i, say q_i, and then recurse on the partial sink spanner procedure on q_i and $S_i \setminus \{q_i\}$. In the case that one cone contains more than half of the points, split the points in the cone to two almost equal parts and do the same thing as above. This guarantees that the subproblems half in size.

2.8 Skip-List Spanner

To obtain a spanner with bounded spanner diameter, one can use skip-list spanners as suggested by Arya et al. [4]. The idea is to generalize skip-lists and apply them to the construction of t-spanners.

To construct a t-spanner of V, we construct a sequence of subsets of V, $V = V_0 \supseteq V_1 \supseteq \cdots \supseteq V_k = \emptyset$. To construct V_{i+1}, we flip a fair coin for each element of V_i and then add the point to V_{i+1} if the flip produce heads. The construction ends when the set is empty. Now we construct a t-spanner using the Θ-graph algorithm for each V_i and the union of all these graphs is the skip-list spanner of V.

Theorem 6. *The skip-list spanner is a t-spanner for $V \subset \mathbb{R}^2$ with $\mathcal{O}(kn)$ edges, $\mathcal{O}(\log n)$ spanner diameter and can be constructed in time $\mathcal{O}(kn \log n)$. All the bounds are expected with high probability.*

Implementation. To construct a skip-list spanner, we construct a t-spanner on V using the Θ-graph algorithm. Then for each point in the set we produce a random number between 0 and 10,000 using `random_source` type in LEDA and remove the point if the outcome is less than 5,000. Then again we construct the Θ-graph on the remaining points and we add the generated edges to the previous graph. We continue this procedure until we have no remaining points in the set.

2.9 The Hybrid Algorithms

In [10] it was experimentally shown that the greedy algorithm produced graphs whose size, weight and degree are superior to the graphs produced from the other approaches. However the running time of the greedy algorithm is $\mathcal{O}(n^3 \log n)$. A way to improve the running time while, hopefully, still obtaining the high-quality graphs is to first compute a t^α-spanner $(0 < \alpha < 1)$ $G(V, E)$ of the input set and then compute a $(t^{1-\alpha})$-spanner of $G(V, E)$ using the greedy pruning algorithm (in the experiments we use the improved greedy algorithm). The resulting graph will have dilation at most $t^{1-\alpha} \cdot t^\alpha = t$. The greedy pruning algorithm is identical to the greedy algorithm, but instead of considering the edges in the complete graph the algorithm only considers the edges in E. The time complexity of the implemented greedy pruning is $\mathcal{O}(mn \log n)$, where m is the number of edges in the input graph.

Table 1. Summarizing the known bounds for the algorithms presented in the paper. The entries marked (*) implies that the values are expected with high probability. The entries marked with (†) indicates that the versions implemented in this paper has an additional $\log n$-factor in their running times.

-	Edges	Weight	Degree	Diameter	Time
Greedy-graph	$\mathcal{O}(\frac{n}{t-1})$	$\mathcal{O}(\frac{1}{(t-1)^4} \cdot wt(MST))$	$\mathcal{O}(\frac{1}{t-1})$	$\Theta(n)$	$\mathcal{O}(n^3 \log n)$
Apx. greedy-graph	$\mathcal{O}(\frac{n}{(t-1)^3})$	$\mathcal{O}(\frac{1}{(t-1)^4} \cdot wt(MST))$	$\mathcal{O}(\frac{1}{(t-1)^3})$	$\Theta(n)$	$\mathcal{O}(\frac{n}{(t-1)^7} \log n)^\dagger$
Θ-graph	$\mathcal{O}(n/\theta)$	$\Theta(n \cdot wt(MST))$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(n/\theta \log n)$
O. Θ-graph	$\Theta(n/\theta)$	$\mathcal{O}(n \cdot wt(MST))$	$\mathcal{O}(1/\theta \cdot \log n)$	$\Theta(n)$	$\mathcal{O}(n/\theta \log n)^\dagger$
WSPD-graph	$\Theta(\frac{n}{(t-1)^2})$	$\mathcal{O}(\log n \cdot wt(MST))$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}((\frac{t}{t-1})^2 n + n \log n)$
Sink-spanner	$\Theta(n/\theta)$	$\mathcal{O}(n \cdot wt(MST))$	$\mathcal{O}(\frac{1}{(t-1)^2})$	$\Theta(n)$	$\mathcal{O}(n/\theta \log n)$
Skip-list spanner	$\Theta(n/\theta)^*$	$\Theta(n \cdot wt(MST))^*$	$\Theta(n)$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(n/\theta \log n)^*$

3 Experimental Results

In this section we discuss the experimental results in more detail by considering the running times of the algorithms. The experiments were done on point sets ranging from 100 to 10,000 points with four different distributions:

– uniform distribution,
– normal distribution with mean 500 and deviation 100,
– gamma distribution with shape parameter 0.75, and
– $\sqrt{n}$ uniformly distributed unit squares with $\sqrt{n}$ uniformly distributed points.

Below we will focus on the uniform distribution and the cluster distribution. A discussion considering all the distributions will be available in the full version of the paper together with a comparison of all the graphs produced by the algorithms. To avoid the effect of specific instances, we ran the algorithms on many different instances and took the average of the results.

3.1 Implementation Details

The algorithms were implemented in C++ using the LEDA 5.01 library. In the cases when LEDA did not contain the required data structure needed for the algorithms, we implemented it ourselves.

The experiments were performed on an AMD Opteron 250 (2.4 GHz), 1GB L2 cache and 4GB RAM. The OS was Fedora 3.4 and it used g++ 3.4.4 for compiling the program using `-O2` option. All sample points sets were generated by NEWRAN03 [9] pseudo random number generator.

3.2 Uniform Distribution

The running times of all the implemented algorithms for $t = 2$ and $t = 1.1$ are depicted in Fig. 1. As the theoretical bounds suggest the original greedy algorithm has the highest time complexity of the implemented algorithms and it shows clearly in the experiments. However, the suggested improvement, the improved greedy algorithm, performed very well in the experimental study and the results corroborate our conjecture that only a linear number of shortest path queries are needed. Figure 1c shows the number of shortest path queries performed by the algorithm for $t = 2$. As an example of the improved running time we constructed a greedy 2-spanner on a set of 4K uniformly distributed points; the original algorithm required 12K seconds while the improved algorithm needed roughly 34 seconds. The improved greedy algorithm performed approximately 13K shortest path queries while the original algorithm performs roughly 8 million queries. Using the improved algorithm we are able to construct greedy graphs for much larger points sets than earlier. For instance for a set of 10K points we can construct 2-spanner greedy graph in about 300 seconds. Figures 3 and 4 illustrates the quality of the obtained graphs using different quality measures.

Based on the experiments, the running time of the improved greedy algorithm is comparable to the running times of the hybrid algorithms using $\alpha = 0.5$ for $t = 2$ and it performs even better for smaller values of t, see Fig. 1 for a comparison. Thus, if high quality networks is a priority the improved greedy algorithm is probably the best choice, especially for small values of t, see Fig. 3 and Fig. 4. Note that the improved greedy algorithm generates the same graph as the original greedy algorithm.

For the hybrid approach, three different values of α were used, $0.1, 0.5$ and 0.9, and the results can be seen in Fig. 2a. By increasing α, less time is used to build the initial graph while more time is needed for the pruning process, see Fig. 2b. However, in [10] it was clearly shown that the decrease in speed gave a better quality network, i.e. small size, low degree and low weight.

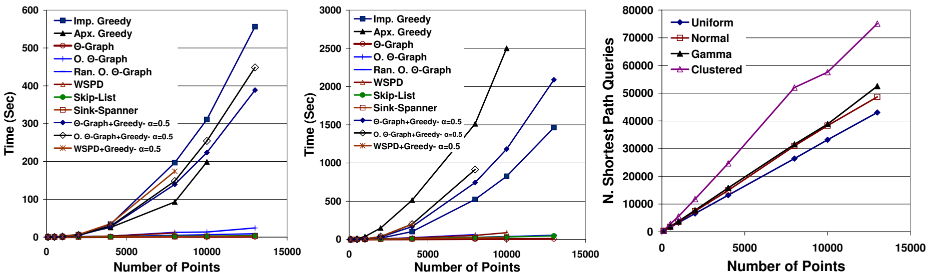


Fig. 1. Comparing the running times of the implemented algorithms for Uniform dist., (a) $t = 2$ and (b) $t = 1.1$. Note the difference between the approximate greedy algorithm and the improved greedy algorithm for the two values of t. (c) Number of shortest paths queries performed by the improved greedy algorithm with $t = 2$.

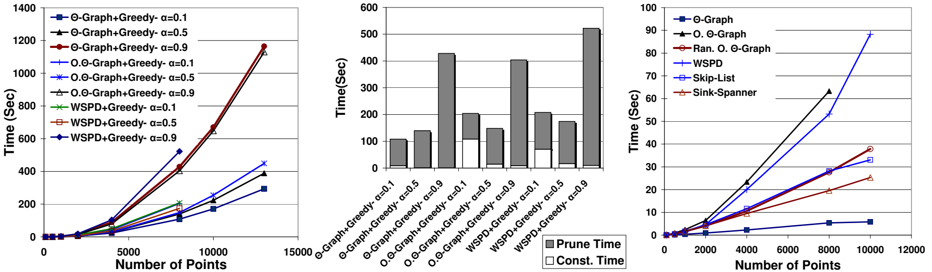


Fig. 2. (a) The performance of the hybrid algorithms for different values of α (Uniform distr. and $t = 2$). (b) Comparing the construction time of the initial graph with the pruning time (Uniform distr., $t = 2$ and $n = 8000$). (c) The running time for the $\mathcal{O}(n \log n)$-time algorithms in the experiments for Uniform distr. with $t = 1.1$.

The remaining algorithms all have a theoretical $\mathcal{O}(n \log^2 n)$, or even $\mathcal{O}(n \log n)$, time complexity. However, the difference in their actual running times is quite substantial and for some a bit surprising. The Θ-graph algorithm is superior to the others with respect to the running time. For sets containing 10K points and for t between 1.5 and 2 the Θ-graph was constructed in less than two seconds. For $t = 1.1$ the running time increased to approximately 6.5 seconds, which is to be expected since its running time is highly dependent on the value of $1/(t-1)^2$. The fastest algorithms after the Θ-graph construction were the sink-spanner algorithm, the skip-list spanner algorithm and the random Ordered Θ-graph algorithm which basically all are modified Θ-graph algorithms. Again for 10K points they required a couple of seconds for $t = 2$ and approximately half a minute for $t = 1.1$. These three algorithms almost show a linear time behavior in our experiments, see Fig. 2c.

For uniform sets the ordered Θ-graph algorithm and the WSPD algorithm clearly show a superlinear behavior but they are still fast enough to handle 8K points with $t = 1.1$ in roughly one minute. For smaller values of t and larger point sets the ordered Θ-graph algorithm ran into memory problems. The simplified

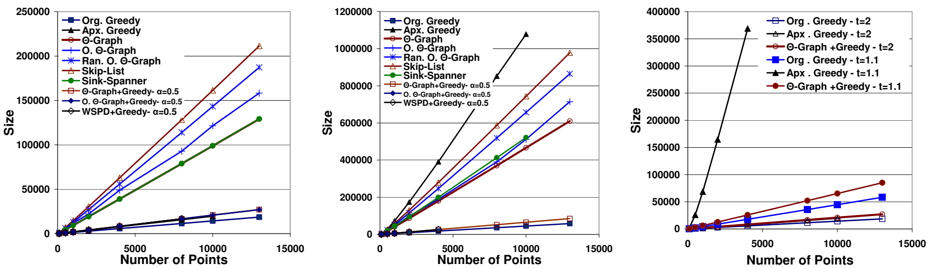


Fig. 3. Comparing the size of the produced graphs for uniform point sets and (a) $t = 2$, and (b) $t = 1.1$. (c) Comparing the size of the high-quality spanners using different values of t.

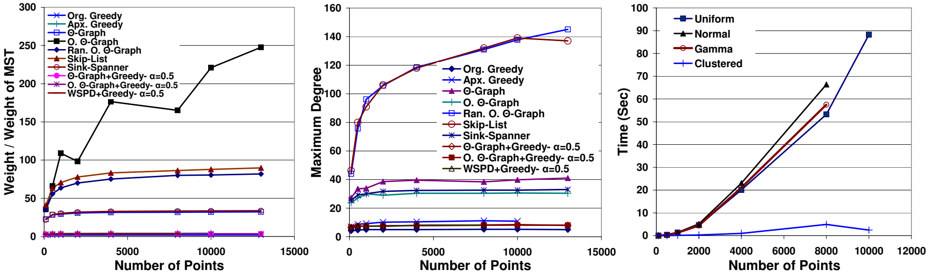


Fig. 4. (a) The weight of the produced graphs for uniform point sets and $t = 2$. (b) The average maximum degree of the produced graphs for uniform distributions and $t = 2$. (c) The running time for the WSPD algorithm for $t = 1.1$ for different distributions.

version that we implemented uses $\Omega(\frac{2\pi}{\theta} n \log n)$ space (instead of $\Omega(\frac{2\pi}{\theta} n)$ space) and for small values of t and large values of n this function grows rapidly.

The approximate greedy algorithm works well for large values of t. For $t = 2$ the running time is comparable to the fastest hybrid algorithms but the produced graphs can be shown to have slightly better quality. When t decreases the running time of the algorithm deteriorates rapidly and for $t = 1.1$ the algorithm performs even worse than the improved algorithm which is conjectured to have a running time of $\mathcal{O}(n^2 \log n)$ (see Fig. 1). The reason is that the approximate greedy approximates the greedy algorithm in two steps; first the complete graph is approximated using a dense t'-spanner G' and then the shortest path queries in G' are approximated using a cluster graph H. This works well for large values of t, and in theory for any constant, however, in practice t becomes too small at some point and the error when doing the approximation becomes too large. As a result the initial graph G' will be very dense (but still linear in n) and the approximation factor used for the approximate shortest path query will be so small that it is equivalent to the exact shortest path query in many cases.

Finally, the produced graphs most often contain many "redundant" edges, i.e., edges that could be removed while still keeping the dilation bounded by t. Table 2 clearly shows this, e.g., the skip-list spanner, sink-spanner and Θ-graph all produce spanners of dilation approximately 1.2 in the case when $t = 2$.

3.3 Clustered Distributions

Most of the algorithms perform slightly better on the clustered point sets, except the WSPD-algorithm and the approximate greedy algorithm which both show a considerable improvement. For example, to construct a 2-spanner on a uniformly distributed set which contains 8K points, the WSPD algorithm needs roughly 11 seconds while the corresponding running time for the clustered set is about 1.6 seconds. For $t = 1.1$ the improvement is even bigger; 88 seconds compared to 2.5 seconds, see Fig. 4c. The WSPD algorithm was expected to perform slightly better for clustered sets since it uses a clustering approach, but the improvement was greater than predicted. Especially for small values of t the

Table 2. The maximum dilation of graphs generated by different algorithms

Maximum Dilation (Uniform distribution)										
	$t = 2$					$t = 1.1$				
n	500	1000	2000	4000	8000	500	1000	2000	4000	8000
Original greedy	1.99	2	2	2	2	1.1	1.1	1.1	1.1	1.1
Improved greedy	1.99	2	2	2	2	1.1	1.1	1.1	1.1	1.1
Approximate greedy	1.68	1.68	1.68	1.68	1.68	1.07	1.07	1.07	1.07	
Θ-graph	1.18	1.2	1.21	1.23	1.22	1.02	1.02	1.03	1.03	1.03
O. Θ-graph	1.37	1.4	1.43	1.46	1.47	1.06	1.07	1.07	1.07	1.07
Random O. Θ-graph	1.35	1.38	1.42	1.41	1.47	1.06	1.06	1.07	1.07	1.07
WSPD-graph	1.35	1.39	1.44	1.49	1.47	1.04	1.04	1.05	1.05	1.05
Skip-list	1.17	1.18	1.21	1.21	1.21	1.02	1.02	1.03	1.03	1.03
Sink-spanner	1.19	1.19	1.21	1.23	1.23	1.03	1.03	1.03	1.03	1.04

algorithm performs better, it is even comparable to the Θ-graph algorithm for the clustered set with 10K points and $t = 1.1$. A similar observation can be made for the approximate greedy where the corresponding running times for $t = 1.1$ and $n = 8K$ are 1500 seconds and 128 seconds. As for the WSPD-approach the approximate greedy algorithm also uses a clustering approach however the main gain comes from the fact that the algorithm does not process any edges in the initial graph G' of length at most D/n (they are just added to the partial spanner graph), where D is the diameter of the point set. In the clustered case there will be many such edges and thus only "long" edges has to be processed.

An interesting observation that can be seen in Fig. 1c is that the number of shortest path queries performed by the improved greedy algorithm in uniformly distributed sets is considerably smaller than for the clustered points set, while the running time is almost the same. Consider the case when the input contains 10K points. The number of shortest path queries performed on the uniform set is approximately 33K while it is about 57K for the clustered set. The number of clusters is 100, with 100 points per cluster. From the experiments it follows that the number of shortest-path queries performed between two points within the same cluster of uniformly distributed points is approximately 300. Since there are 100 clusters the number of shortest path queries needed for the "intra-cluster" edges in the clustered set is approximately 30K. These queries are all performed on very small graphs and are therefore processed extremely fast. Next approximately 27K "inter-cluster" queries are performed. We believe that the smaller number of "inter-cluster" queries together with the fact that the 2-spanner of the clustered set is slightly smaller than for the uniform set explains why the running times for the two different distributions are almost identical.

4 Concluding Remarks and Acknowledgements

We studied the running time of the most common construction algorithm for t-spanners. In addition to the algorithms presented in [10] we also tested

sink-spanners, skip-list spanners and, most importantly, the approximate greedy spanner. Unfortunately, the approximate greedy algorithm performs worse than expected in most cases, even though the theoretical bounds are very good.

In general the Θ-graph construction algorithm is the fastest algorithm, however if it is important to obtain a high quality network then the improved greedy algorithms seems to be the most suitable choice. The main question that remains to be answered experimentally is the dependency on the number of dimensions, i.e. how the algorithms and the quality of the produced graphs depends on the number of dimensions.

The authors would like to thank the anonymous reviewers for comments on a earlier version of this paper.

References

[1] Althöfer, I., Das, G., Dobkin, D.P., Joseph, D., Soares, J.: On sparse spanners of weighted graphs. Discrete & Computational Geometry 9(1), 81–100 (1993)

[2] Arya, S., Das, G., Mount, D.M., Salowe, J.S., Smid, M.: Euclidean spanners: short, thin, and lanky. In: Proc. 27th ACM Symposium on Theory of Computing, pp. 489–498. ACM Press, New York (1995)

[3] Arya, S., Mount, D.M., Smid, M.: Randomized and deterministic algorithms for geometric spanners of small diameter. In: Proc. 35th IEEE Symposium on Foundations of Computer Science, pp. 703–712 (1994)

[4] Arya, S., Mount, D.M., Smid, M.: Dynamic algorithms for geometric spanners of small diameter: Randomized solutions. Computational Geometry: Theory and Applications 13(2), 91–107 (1999)

[5] Bose, P., Gudmundsson, J., Morin, P.: Ordered theta graphs. Computational Geometry: Theory and Applications 28, 11–18 (2004)

[6] Callahan, P.B., Kosaraju, S.R.: A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. Journal of the ACM 42, 67–90 (1995)

[7] Clarkson, K.L.: Approximation algorithms for shortest path motion planning. In: Proceedings of the 19th ACM Symposium on the Theory of Computing, pp. 56–65 (1987)

[8] Das, G., Narasimhan, G.: A fast algorithm for constructing sparse Euclidean spanners. International Journal of Computational Geometry and Applications 7, 297–315 (1997)

[9] Davis, R.B.: http://www.robertnz.net/nr03doc.htm (2005)

[10] Farshi, M., Gudmundsson, J.: Experimental study of geometric t-spanners. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 556–567. Springer, Heidelberg (2005)

[11] Gudmundsson, J., Levcopoulos, C., Narasimhan, G.: Improved greedy algorithms for constructing sparse geometric spanners. SIAM Journal of Computing 31(5), 1479–1500 (2002)

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

[13] Keil, J.M., Gutwin, C.A.: Classes of graphs which approximate the complete Euclidean graph. Discrete and Computational Geometry 7, 13–28 (1992)

[14] Li, X.-Y.: Applications of computational geometry in wireless ad hoc networks. In: Cheng, X.-Z., Huang, X., Du, D.-Z. (eds.) Ad Hoc Wireless Networking, Kluwer Academic Publishers, Dordrecht (2003)
[15] Narasimhan, G., Smid, M.: Geometric spanner networks. Cambridge University Press, Cambridge (2007)
[16] Navarro, G., Paredes, R.: Practical construction of metric t-spanners. In: Proc. 5th Workshop on Algorithm Engineering and Experiments, pp. 69–81. SIAM Press (2003)
[17] Rao, S., Smith, W.D.: Approximating geometrical graphs via spanners and banyans. In: Proc. 30th ACM Symposium on the Theory of Computing, pp. 540–550. ACM, New York (1998)
[18] Sigurd, M., Zachariasen, M.: Construction of minimum-weight spanners. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, Springer, Heidelberg (2004)

Vertex Cover Approximations on Random Graphs

Eyjolfur Asgeirsson[1] and Cliff Stein[2]

[1] Reykjavik University, Reykjavik, Iceland
eyjo@ru.is
[2] Department of IEOR, Columbia University, New York, NY
cliff@ieor.columbia.edu

Abstract. The vertex cover problem is a classical NP-complete problem for which the best worst-case approximation ratio is $2 - o(1)$. In this paper, we use a collection of simple graph transformations, each of which guarantees an approximation ratio of $\frac{3}{2}$, to find approximate vertex covers for a large collection of randomly generated graphs. These reductions are extremely fast and even though they, by themselves are not guaranteed to find a vertex cover, we manage to find a $\frac{3}{2}$-approximate vertex cover for almost every single random graph we generate.

1 Introduction

In 1972, a year after Cook [2] formalized the notion of NP-completeness and proved that the boolean satisfiability problem is NP-complete, Karp [11] showed that 21 diverse problems from graph theory and combinatorics are NP-complete. One problem from this set of 21 NP-complete problems is the vertex cover problem. A vertex cover of a graph $G = (V, E)$ is a subset of the vertices, $C \subseteq V$, such that each edge $e \in E$ has at least one endpoint in C. The objective is to minimize the size of the vertex cover.

The best known approximation ratio for vertex cover is $2 - \Theta(\frac{1}{\sqrt{\log n}})$. This approximation ratio is achieved by an algorithm by Karakostas [10]. It is NP-hard to approximate minimum vertex cover within any factor smaller than 1.36 [5]. A simple greedy algorithm gives a 2-approximation for the vertex cover problem [6] and several people conjecture that there does not exist an algorithm with a fixed approximation ratio better than 2 [9]. Other work on vertex cover includes [4,12,8].

In 2005, Asgeirsson and Stein [1] introduced the idea of finding an approximate vertex cover using simple graph reductions. These reductions are not guaranteed to find a vertex cover, but if successful give a $\frac{3}{2}$-approximate vertex cover (and usually one that is significantly better). Asgeirsson and Stein showed that by using the heuristic, they could find a $\frac{3}{2}$-approximate vertex cover for every single graph that they tried, which included every graph that they found in the literature on vertex cover.

In this paper, we will extend and improve the results of Asgeirsson and Stein in several ways. First, we will introduce a new graph reduction, which we call

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 285–296, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the *almost bipartite* reduction. We will then analyze it, and relate it to the other graph reductions introduced in [1]. In particular, we will show that there are classes of graphs for which the heuristics of [1] do not find a vertex cover, but the addition of the *almost bipartite* reduction allows us to find a vertex cover. Second, we will extend the class of graphs analyzed to include *random graphs*. The random graphs that we use are based on the simple random graph model by Erdös and Rényi [7], where we start with a fixed set of vertices and add edges to the graph based on a edge probability parameter. Finally, we give a more detailed analysis than [1], and analyze the performance of each graph reduction with respect to various factors, such as the density of the graph and the number of vertices. We will also show that the choice of reductions depends on the edge probabilities, and that for certain probabilities, the almost-bipartite reductions is very useful.

The graph reductions do not guarantee that we will find an approximate vertex cover, but in our experiments we managed to find a $\frac{3}{2}$-approximate vertex cover for almost every single graph we tried. For the graphs where we know the size of the minimum vertex cover, the actual approximation ratio was usually much lower than $\frac{3}{2}$; in many cases the vertex cover we found was either optimal or very close to optimal.

2 Graph Reductions

Asgeirsson and Stein [1] introduced a heuristic, based on simple graph reductions, that tries to find a $\frac{3}{2}$-approximate vertex cover by breaking the graph into smaller and easier subproblems. The graph reductions focus on special graph structures and their goal is to remove a set V' of vertices and a set E' of edges from the graph by finding a vertex cover for the subgraph induced by the set V'. The reductions are designed so that when we combine the vertex covers for all the subgraphs, we get an approximate vertex cover for the original graph. The approximation ratio of the vertex cover is equal to maximum approximation ratio over all the reductions that are used. Unless otherwise stated, we will only use reductions with approximation ratio no more than $\frac{3}{2}$, so the resulting vertex cover for the original graph, if we succeed in finding one, will have an approximation ratio of $\frac{3}{2}$. Before we list the graph reductions, we need the following definitions [1]:

Definition 1. *An* **optimal graph reduction** *is a mapping from a graph $G = (V, E)$ to a graph $G' = (V', E')$ with the property that if we have an optimal vertex cover VC'_{opt} for G' then we can create an optimal vertex cover for the original graph G from VC'_{opt} and from the graph reductions that we performed on the graphs.*

Definition 2. *A* **ρ-approximating graph reduction** *is a mapping from a graph $G = (V, E)$ to a graph $G' = (V', E')$ such that if we have an optimal vertex cover VC' for G' then we can create a ρ-approximate vertex cover for G from VC' and from the graph reductions that we performed on the graph.*

We will use both optimal graph reductions and approximate graph reductions. Most of the approximate reductions have approximation ratio $\rho \leq \frac{3}{2}$, but we will introduce reductions that have higher approximation ratios. An operation we will use extensively is a *vertex contraction*.

Definition 3. *The* **contraction of a set of vertices** $v_1, \ldots, v_k$ *to a new vertex* v *is an operation where we replace the vertices* $v_1, \ldots, v_k$ *with a new vertex* v*, delete all edges between removed vertices and replace each edge* (v_i, u) *with an edge* (v, u)*. The set of vertices adjacent to* v *is the union of the vertices that were adjacent to* $v_1, \ldots, v_k$*. If we use the mapping to* v *to find an approximate vertex cover for the original graph we call* v *an* **approximated vertex***.*

When we perform a vertex contraction, we replace multiple edges that might appear with a single edge and encode information about the contracted vertices and adjacent edges so that we can recreate them later to get the original graph.

Most of the graph reductions that we use are defined in [1]. For completeness, we list the graph reductions here along with a brief overview. The name we use for each reduction refers to the graph structure that the reduction uses to attack the graph. Since we are interested in finding an approximate vertex cover, and each reduction we use has a particular approximation ratio, we group the reductions into optimal and approximate graph reductions, according to Definitions 1 and 2.

Optimal Graph Reductions

- **Degree-0 vertices:** Removes a single vertex of degree-0 and no edges.
- **Degree-1 vertices:** There is an optimal cover that includes the neighbor and not the degree-1 vertex. This reduction removes the degree-1 vertex and its neighbor along with the edge between them. Edges connected to the neighbor are also removed.
- **Degree-2 vertices with adjacent neighbors:** There is an optimal vertex cover that includes both neighbors and not the degree-2 vertex. We remove the degree-2 vertex and both its neighbors along with all edges adjacent to these three vertices.
- **Degree-2 vertices with non-adjacent neighbors:** Contracts the degree-2 vertex and its neighbors into a single vertex for a new graph G'. Removes at least two edges. The contracted vertex is not approximated. If the optimal cover for the new graph includes the contracted vertex, then the optimal cover for the original graph includes both neighbors and not the degree-2 vertex. If the optimal cover for G' does not include the contracted vertex then the optimal vertex cover for the original graph includes the degree-2 vertex and not the neighbors.
- **Extended Network Flow:** Removes all vertices corresponding to non-half variables in the optimal solution to the LP-relaxation of the vertex cover problem. We also try to fix each variable equal to 1 and resolve in order to get an optimal LP solution with more non-half variables. If the optimal solution with a variable fixed as 1 is equal to the optimal solution with no fixed variables, we get a new optimal solution with more non-half variables that we can remove.

Approximate Graph Reductions

- **Triangle elimination:** Removes the three vertices in the triangle and all edges connected to these vertices. Any cover must use at least 2 of the vertices, so by using all 3 we get a $\frac{3}{2}$-approximation.
- **Degree-3 vertices:** Contracts the degree-3 vertex and its neighbor into a single approximated vertex. Removes at least three edges. This is a $\frac{3}{2}$-approximation.
- **Four-cycles:** Contracts the four vertices from a chordless cycle of length four into two approximated vertices. Removes at least three edges. The four-cycle reduction is a $\frac{4}{3}$-approximate graph reduction.
- **Five-cycles:** Removes the five vertices from a cycle of length five and all edges connected to these vertices. The five-cycle reduction is a $\frac{5}{3}$-approximate reduction. We will only use this reduction if we cannot find a $\frac{3}{2}$-approximate vertex cover by using the other reductions.
- **Six-cycles:** Contracts the six vertices in a chordless cycle of length six into two approximated vertices and removes at least five edges. The approximation ratio of the 6-cycle reduction is $\frac{3}{2}$.

2.1 New Graph Reduction: Almost-Bipartite

When we are trying to solve NP-hard problems it is often helpful to look for special cases. For many problems, while the general problem is NP-hard, there are special cases that yield an optimal solution in polynomial time. One such example is the vertex cover for bipartite graphs. The vertex cover problem is well known to be NP-hard for general graphs but for the special case of bipartite graphs, we can find an optimal vertex cover in polynomial time using maximum matching. It is also well known that the size of the minimum vertex cover for bipartite graphs is equal to the size of the maximum matching [3].

We can use the fact that it is easy to find an optimal vertex cover for a bipartite graph to try to find approximate vertex covers for general graphs. If we have a lower bound of k for the optimal vertex cover, we know that any vertex cover of size no more than pk is a $p-$approximate vertex cover.

The **almost bipartite** method is as follows: We partition the vertices of the graph into three sets, A, B and C, such that the subgraph induced by the first two sets of vertices, A and B, will be bipartite. The last set, C, will include all vertices that violate the bipartite property if they are added to the subgraph, i.e. each vertex in the set C has neighbors in both sets A and B.

Claim. Let $G = (V, E)$ be a graph. Assume we partition V into three sets, A, B and C, with the property that the subgraph induced by the sets A and B is bipartite. Let G_{AB} be the bipartite subgraph induced by the sets A and B and let $\mathrm{VC}^*_{G_{AB}}$ be an optimal vertex cover for G_{AB}. Also let VC_{LB} be a lower bound on the size of the optimal vertex cover for the original graph, and let p be the approximation ratio that we want to achieve. Then, if $|\mathrm{VC}^*_{G_{AB}}| + |C| \leq p\mathrm{VC}_{LB}$, the set $\mathrm{VC}^*_{G_{AB}} \cup C$ is a p-approximate vertex cover for G.

Proof. The vertex cover $\mathrm{VC}^*_{G_{AB}}$ covers all edges in G_{AB} and all other edges in G are covered by the set C, so $\mathrm{VC}^*_{G_{AB}} \cup C$ is a feasible vertex cover for G. If VC_{LB} is an lower bound on the optimal vertex cover for G, then any feasible vertex cover whose size is smaller than $p\mathrm{VC}_{LB}$ is a p-approximate vertex cover for G. We know that $\mathrm{VC}_{LB} \leq |\mathrm{VC}^*|$, where VC^* is an optimal vertex cover for G. If the sum of the size of the set C, and the size of the optimal vertex cover on the bipartite subgraph G_{AB} is smaller than $p\mathrm{VC}_{LB}$, then $|\mathrm{VC}^*_{G_{AB}}| + |C| \leq p\mathrm{VC}_{LB} \leq p|\mathrm{VC}^*|$ and $\mathrm{VC}^*_{G_{AB}} \cup C$ is a p-approximate vertex cover for G. □

In our experiments, the approximation ratio p is usually equal to 3/2. The almost-bipartite method succeeds if the size of the optimal vertex cover on the subgraph induced by the sets A and B plus the size of set C is less than the upper limit on the size of the approximate vertex cover, i.e. if $|\mathrm{VC}^*_{G_{AB}}| + |C| \leq p\mathrm{VC}_{LB}$.

The problem of finding a maximum sized bipartite subgraph in a graph is NP-hard [4]. Finding a maximum sized bipartite subgraph is a special case of the problem of finding a maximum induced subgraph with property P, where property P is hereditary and non-trivial. A property P of graph G is hereditary if every subgraph of G also satisfies P. In our case, the property P is the bipartite property. The problem of finding a maximum induced subgraph with property P is approximable within $O(\frac{n}{log(n)})$ where n is the number of vertices. However, the problem is not approximable within n^ϵ for some $\epsilon > 0$ unless $P = NP$.

Therefore, we settled for heuristics to find the almost-bipartite subgraph. We tested a few different implementations of the almost-bipartite method. However, the comparison between these methods indicated that there was not a large difference between the various partitioning methods. We settled on a greedy method in which we start by adding a single vertex to set A and then focus on each set in turn, switching between sets only when we have processed every vertex in the current set of vertices. A vertex is processed by iterating through all its neighbors and placing them in the appropriate sets. The unprocessed vertices in each set are selected in the order in which they are added to the set. If there are unprocessed vertices after we process all vertices in the sets A and B, we add a randomly selected unprocessed vertex to set A and continue as before. If the algorithm fails, we try starting from another vertex until we either find an approximate vertex cover or until we have tried starting from all the vertices, in which case the algorithm fails.

It is somewhat tricky to combine the almost bipartite method with the other reductions since the almost bipartite method cannot work on graphs that contain approximated vertices. The almost bipartite method also needs a good lower bound on the size of the optimal vertex cover in order to work. However, if we can overcome these obstacles then the almost-bipartite reduction is very powerful because when it succeeds, it completely solves the graph.

2.2 The Automated Order of Reductions

The automated order of the graph reductions is the same order that was suggested in [1], with the addition of the almost bipartite method. We run the

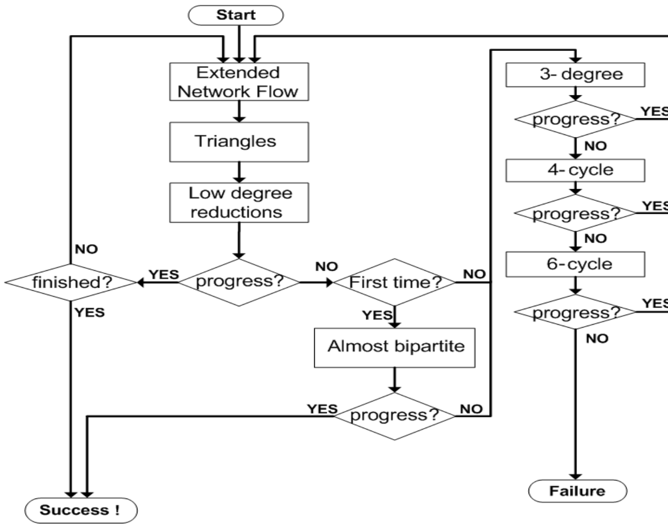


Fig. 1. A flowchart showing the automated order of the reductions

extended network flow method, triangle elimination and low-degree in a loop until we find a solution or no improvements are made during an iteration. If the graph is not empty and we have not created any approximated vertices, we try the almost bipartite reduction. If the almost bipartite method does not solve the graph, we try the 3-degree, 4-cycle and 6-cycle reductions, stopping as soon as any one of them makes some progress and return to the original loop. The stopping criteria is either having processed all the vertices from the graph which gives us a vertex cover, or running 3-degree, 4-cycle and 6-cycle without any improvements. In that case we stop and must use some other methods, such as branch-and-bound, to get a solution. If we find a solution then the final step in the algorithm is to use a simple greedy algorithm to eliminate unnecessary vertices from the cover. In our experiments we almost never had to resort to branch-and-bound, our algorithm managed to solve almost every single graph we found. The flowchart for the automated order is shown in Figure 1. The 5-cycle reduction is not included in Figure 1 since its approximation ratio is higher and we only use it if we cannot find a $\frac{3}{2}$-approximate vertex cover by using the other reductions.

3 Experiments on Random Graphs

The random graphs we generated are based on the simple random graph model by Erdös and Rényi [7]. We start with fixed set of vertices V where $|V| = n$. First we randomly select an edge density parameter ρ such that $0 < \rho < 1$. Then for each possible edge e, we add the edge with probability ρ, i.e. for any

possible edge, we generate a random number between 0 and 1. If this randomly generated number is less than ρ, we add the edge to the graph, while if the randomly generated number is greater than ρ the edge is not included in the graph. We try this once for every possible edge, i.e. for any pair of vertices $v_i, v_j \in V$.

We used four different sets of vertices, with 200, 500, 1000 and 10,000 vertices. We generated 20,000,000 random graphs on 200 vertices, 1,000,000 random graphs on 500 vertices and 100,000 random graphs on 1000 vertices. Finally we generated 2000 sparse random graphs on 10,000 vertices. By using our reductions, we managed to find $\frac{3}{2}$-approximate vertex cover for every randomly generated graph with 1000 vertices or less. Because we solved every graph with at most 1000 vertices, we will focus on the performance of each reduction and show how effective each reduction is for the smaller random graphs, based on the edge probability. The large graphs, with 10,000 vertices, are a special case since the graphs are all sparse and we only have a relatively small collection. We will analyze the large graphs separately.

3.1 Using the Almost Bipartite Reduction

The almost bipartite method differs from our other reductions because it is guaranteed to finish the graph if it works, and the graph needs to fulfill specific conditions in order for us to use it. We therefore wanted to understand how the almost bipartite method relates to the other reductions and how important it is when we are trying to find approximate vertex covers. The majority of the graphs that we generated were solved by using only the triangle elimination, the extended network flow method and the low degree reductions, but there were some graphs that could not be solved without using the almost bipartite method. Figure 2 shows the fraction of graphs that were solved as a function of

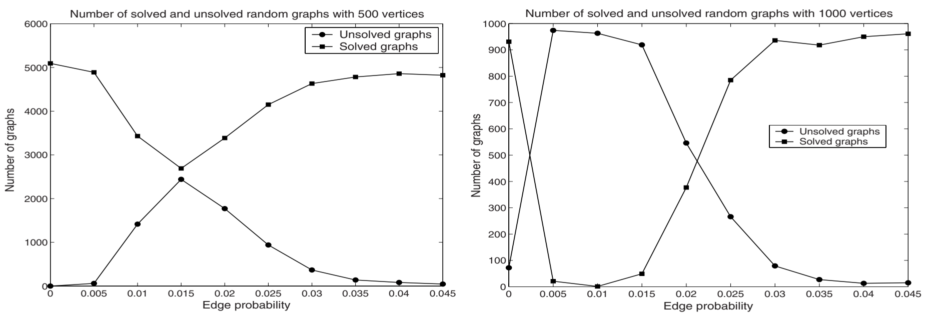


Fig. 2. The number of random graphs that were solved without using the almost bipartite method and the number of graphs that could not be solved without the almost bipartite method. The first graph shows the results for randomly generated graphs on 500 vertices and the second graphs shows the results for random graphs with 1000 vertices.

the edge probability. In Figure 2 we see that the graphs that were not solved without the almost bipartite method all have very similar edge probabilities. For random graphs on 1000 vertices, using an edge probability between 0.005 and 0.015 almost guarantees that the resulting random graph cannot be solved without the almost bipartite method. It is more difficult to generate random graphs with 500 vertices that cannot be solved without the almost bipartite method, since only about one out of every thirty graphs with edge probability between 0.03 and 0.045 cannot be solved without using the almost bipartite method.

3.2 Triangle Elimination

The triangle elimination is by far the most powerful reduction in our arsenal. It is responsible for eliminating the largest number of both vertices and edges. The triangle elimination works extremely well when the graphs are dense. When the edge probability is more than 0.1, the triangle elimination usually removes more than 95 percent of both vertices and edges. The result is that, for finding $\frac{3}{2}$-approximate vertex cover, the only interesting random graphs are the graphs that are very sparse. For the dense random graphs, the triangle elimination is simply too effective for those instances to be interesting. Figure 3 shows the fraction of vertices that the triangle elimination removes from the graphs when we use vertex sets of size 500 and 1000. The graphs use a single point for every graph solved so we see how the distribution in the fraction of removed vertices changes with the edge probability and the number of vertices in the graph. By having more vertices, we decrease the variance of the fraction of removed vertices, which indicates that the effect of the reductions gets more consistent as the random graphs get larger.

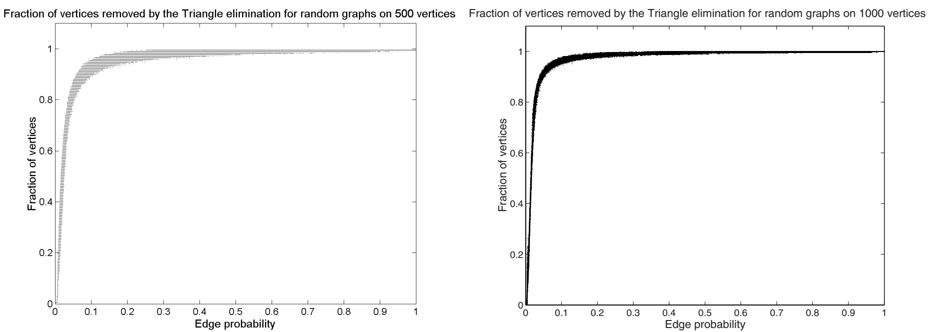


Fig. 3. The fraction of vertices removed by the triangle elimination. The first graph shows the results for randomly generated graphs on 500 vertices and the second graphs shows the results for random graphs with 1000 vertices. The graph show a single point for every solved graph, so the first graph has 1,000,000 points while the second uses 100,000 points.

3.3 Extended Network Flow and Low Degree

The extended network flow method and the low degree eliminations combine to find approximate vertex cover when the graphs are very sparse. When we look at the success of these methods we first need to look at the combined effort since any comparison must take into account which method is used first when we automate the graph reductions to find approximate covers. In our setup, we start by using the extended network flow method, then we use triangle elimination and finally the low degree reductions. This means that if the initial graph is very sparse, the extended network flow method will finish it off easily, while the low degree methods would probably have done the same thing if they were given the chance. Also, by using the low degree methods after we use the triangle elimination, we grant the low degree methods a license to mop up any vertices that remain after the triangle elimination has removed the majority of the graph.

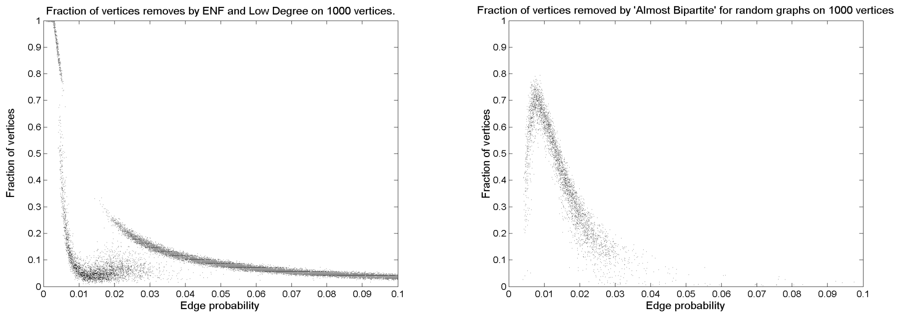


Fig. 4. The fraction of vertices removed from random graphs on 1000 vertices. The first graph shows the combined fraction of vertices removed by the extended network flow method and the low degree methods. The second graph shows the fraction of vertices removed by the almost bipartite method. The graphs are random graphs on 1000 vertices. In both graphs, we focus on edge probabilities less than 0.1.

The first graph in Figure 4 shows that the performance of the low-degree and extended network flow reductions is consistent for every value of edge probability, except when the edge probability is between 0.005 and 0.02 for random graphs with 1000 vertices. This gap is also obvious in Figure 2, since these are the graphs that we could not solve without using the almost bipartite reduction.

3.4 Almost Bipartite Reduction

Figure 3 shows that the results for the triangle elimination are consistent, even for the values of edge probability where the extended network flow and the low degree reductions get into trouble. The fraction of vertices drops sharply once the edge probability falls below a certain threshold, but the graph does not show any jumps or erratic behavior. On the other hand, Figure 4 shows how the extended network flow method and the low degree reductions become

very inconsistent and erratic for certain values of the edge probability, and how the almost bipartite method comes into play when the other reductions are not sufficient to find an approximate vertex cover. We only use the almost bipartite method once the triangle elimination, extended network flow and the low degree reductions have all tried and failed to remove vertices from the graph. When we had to use the almost bipartite method, it worked admirably and found a $\frac{3}{2}$-approximate vertex cover for every single instance of the graphs with 1000 vertices or less.

3.5 Large Sparse Graphs

The reductions that we have introduced do not guarantee that we will find an approximate vertex cover for every single graph. However, we have managed to find approximate vertex covers for every single graph we have introduced so far. When we looked at sparse random graphs with 10,000 vertices, we finally found some graphs for which we couldn't find a $\frac{3}{2}$-approximate vertex cover. By using our graph reductions we reduced the size of the graphs but the reductions were not able to finish off the graphs completely. We generated 2000 graphs with 10,000 vertices and edge probability less than 0.1. We could only solve 1876 of these 2000 graphs using the reductions with approximation ratio less than 3/2. The remaining 124 graphs were solved by adding the five-cycle reduction to the set of reductions, which gives us an approximation ratio of 5/3. We only generated random graphs with edge probability less than 0.1 in order to focus our efforts on the graphs we knew that could be difficult.

The first graph in Figure 5 shows the number of vertices that remain in the graphs when the graph reductions cannot make any more progress. The second graph in Figure 5 shows how many vertices the triangle elimination removed

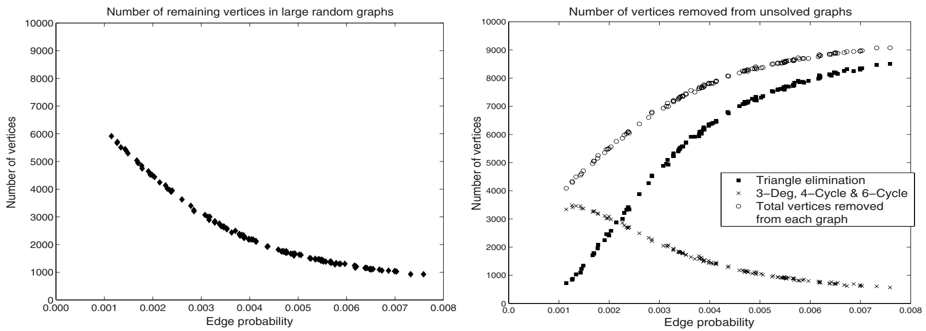


Fig. 5. The graph on the left shows the number of vertices that remain in the graph when our algorithm cannot make any more progress. The graphs are random graphs with 10,000 vertices. The graph on the right shows how many vertices were removed from the graphs in total, how many vertices the triangle elimination removed from the graphs and how many vertices were removed by the 3-degree, 4-cycle and 6-cycle reductions.

from the graphs and how many vertices were removed by the reductions that create approximated vertices. We used three reductions that create approximated vertices, the 3-degree reduction, 4-cycle reduction and 6-cycle reduction. Figure 5 shows that the number of vertices that remain in the graph mostly depends on how effective the triangle elimination is. The triangle elimination is less effective on sparse graphs, when the graphs become denser the triangle elimination is more effective. We only use the 3-degree reduction, 4-cycles and 6-cycle reductions after the triangle elimination has removed all triangles from the graph. The performance of these 3 reductions is therefore dependent on how many vertices the triangle elimination has removed before we start using the other methods.

3.6 Comparison with the Optimal Vertex Cover

We solved some of the random graphs with 200 vertices optimally and compared the size of the optimal vertex cover to the approximate vertex cover that we got from our reductions. The approximation ratio follows a similar pattern as the performance of each reduction, where we find that the edge probabilities of the most interesting graphs are found in a small interval. When the edge probability is too high, the graphs are very dense and the optimal vertex covers are large, which means that any feasible cover is likely to have a good approximation ratio. However, if the edge probability is too small, we can easily solve the graphs optimally using the extended network reduction.

The average approximation ratio over graphs with edge probability ≥ 0.3 is 1.015 with the largest approximation ratio equal to 1.033. When the graphs are very sparse, where the size of the optimal vertex cover is less than 100, the extended network flow method usually manages to find an optimal cover, so the approximation ratio for these graphs is equal to 1. There is however an interval in the edge probability where the graph is not too dense and not too sparse, and in that interval there is more variation in the approximation ratio. The highest approximation ratio is 1.074 and we get that when the edge probability is equal to

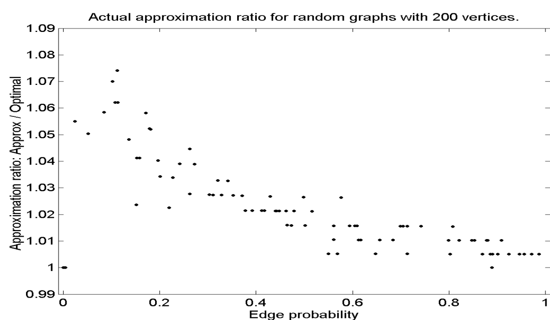


Fig. 6. The ratio of the size of the approximate vertex cover that we get from our reductions over the size of an optimal vertex cover. The graphs are random graphs on 200 vertices.

0.112. The average approximation ratio for graphs with edge probability between 0.05 and 0.2 is equal to 1.052. We see that the largest approximation ratio, 1.074, is still very far from our upper limit of 1.5 and the overall average approximation ratio is equal to 1.023.

4 Conclusions

We used a collection of simple reductions where we allowed reductions that have a worst case approximation ratio of 3/2. Even though these reductions do not guarantee that we will find a solution, we ran these reductions on a wide collection of test problems from every source we could find and by combining them we managed to find an approximate vertex cover for almost every single random graph of the that we generated. The reductions that we use are extremely fast and easily applied, and since the bad examples have a very restrictive structure, these reductions should work well in practice.

References

1. Asgeirsson, E., Stein, C.: Vertex cover approximations: Experiments and observations. WEA, pp. 545–557 (2005)
2. Cook, S.: The complexity of theorem proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing, pp. 151–158 (1971)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT Press, Cambridge, MA, USA (2001)
4. Crescenzi, P., Kann, V.: A compendium of NP optimization problems. http://www.nada.kth.se/theory/problemlist.html
5. Dinur, I., Safra, S.: The importance of being biased. In: Proc. 34th Ann. ACM Symp. on Theory of Comp. pp. 33–42 (2002)
6. Erdös, P., Gallai, T.: On the minimal number of vertices representing the edges of a graph. Publ. Math. Inst. Hungar. Acad. Sci. 6, 181–202 (1961)
7. Erdös, P., Rényi, A.: On random graphs. Publ. Math. Debrecen 6, 290–297 (1959)
8. Halperin, E.: Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. In: Proc. 11th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp. 329–337 (2000)
9. Hochbaum, D.S.: Efficient bounds for the stable set, vertex cover and set packing problems. Discrete Applied Mathematics 6, 243–254 (1983)
10. Karakostas, G.: A better approximation ratio for the vertex cover problem. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, Springer, Heidelberg (2005)
11. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)
12. Monien, B., Speckenmeyer, E.: Ramsey Numbers and an Approximation Algorithm for the Vertex Cover Problem. Acta Inf. 22, 115–123 (1985)

Optimal Edge Deletions for Signed Graph Balancing*

Falk Hüffner, Nadja Betzler, and Rolf Niedermeier

Institut für Informatik, Friedrich-Schiller-Universität Jena,
Ernst-Abbe-Platz 2, D-07743 Jena, Germany
{hueffner,betzler,niedermr}@minet.uni-jena.de

Abstract. The BALANCED SUBGRAPH problem (edge deletion variant) asks for a 2-coloring of a graph that minimizes the inconsistencies with given edge labels. It has applications in social networks, systems biology, and integrated circuit design. We present an exact algorithm for BALANCED SUBGRAPH based on a combination of data reduction rules and a fixed-parameter algorithm. The data reduction is based on finding small separators and a novel gadget construction scheme. The fixed-parameter algorithm is based on iterative compression with a very effective heuristic speedup. Our implementation can solve biological real-world instances exactly for which previously only approximations [DasGupta et al., WEA 2006] were known.

1 Introduction

The concept of balanced signed graphs was first introduced by Harary [11] for the analysis of social networks (see [22] for a bibliography of signed graphs), and has been frequently rediscovered since. In particular, motivated by the mathematical analysis of large-scale biological networks, DasGupta et al. [3] use it to model the concept of "monotone subsystems", under the name of "sign-consistent graphs". A graph $G = (V, E)$ with edges labeled by $h : E \to \{0, 1\}$ is *balanced* if there is a vertex coloring $f : V \to \{0, 1\}$ such that

$$\forall \{u, v\} \in E : h(\{u, v\}) \equiv (f(u) + f(v)) \pmod 2.$$

Put another way, a 0-edge demands that its endpoints have the same color, and a 1-edge demands that they have different colors. Therefore, in the following we use the notations "=-edge" and "≠-edge" instead. The task of decomposing a network into monotone subsystems is then formulated as the graph modification problem BALANCED SUBGRAPH, called UNDIRECTED LABELING PROBLEM by DasGupta et al. [3]. This problem also finds numerous other applications, e. g., in statistical physics and integrated circuit fabrication techniques [2,22].

* Supported by the Deutsche Forschungsgemeinschaft, Emmy Noether research group PIAF (fixed-parameter algorithms, NI 369/4), project PEAL (parameterized complexity and exact algorithms, NI 369/1), and project ITKO (iterative compression for solving hard network problems, NI 369/5).

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 297–310, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Given an undirected graph $G = (V, E)$ with edges labeled by $h : E \to \{=, \neq\}$, the Balanced Subgraph problem is to find a balanced subgraph with maximum number of edges. Balanced Subgraph is a generalization of the NP-hard Maximum Cut problem in graphs. Based on semidefinite programming for Max2SAT [20], DasGupta et al. [3] developed an approximation algorithm that guarantees a solution with at least 87.9% of the number of edges of an optimal solution. They further showed that this approximation factor cannot be improved arbitrarily.

In several applications, one may assume that only a small fraction of the graph edges has to be omitted; therefore, it seems even more attractive to study the formulation as a minimization problem: minimize the number of edges that have to be deleted to make the graph balanced. The special case where all edges are labeled by $\neq$ is the Edge Bipartization problem, which asks for the minimum number of edges to delete to make a graph bipartite. Edge Bipartization, also known as (unweighted) Minimum Uncut, is MaxSNP-hard. Here, the best known approximation algorithm finds in polynomial time a solution of size $O(k \log k)$, where k is the size of an optimal solution [1]. It has been conjectured that it is NP-hard to improve the $\log k$-factor to a constant [15]. Hence, with respect to practical applications using a polynomial-time approximation algorithm of the minimization version seems of hardly any help.

By way of contrast, we have good news from the field of fixed-parameter algorithmics [4,6,16]. A problem is called fixed-parameter tractable with respect to a parameter k if an instance of size n can be solved in $f(k) \cdot n^{O(1)}$ time, where f is an arbitrary function depending only on k. It is known that Edge Bipartization is fixed-parameter tractable with respect to the parameter k as defined above. More specifically, there is an algorithm exactly solving Edge Bipartization in $O(2^k \cdot m^2)$ time (m denoting the number of graph edges) [8]. For relatively small parameter values k, this opens the way to a viable and efficient alternative to employing approximation algorithms.

In this work, we observe that the Balanced Subgraph problem easily reduces to the Edge Bipartization problem and can thus be solved in $O(2^k \cdot m^2)$ time, where k is the number of edges to delete (Sect. 3). Because this (so far only theoretically analyzed) fixed-parameter algorithm in its pure version is still not fast enough to cope with the given real-world instances, we present several tricks and techniques to tremendously speed up its running time in experiments. In this context, our main contribution is a data reduction scheme based on graph separators (Sect. 2). Along with this, we develop a gadget construction that is interesting on its own and deserves further studies from a practical as well as a theoretical side. In addition, directly manipulating the fixed-parameter algorithm [8], we found a speed-up trick that alone reduces the running time of this algorithm in our experiments from days to few seconds (Sect. 4). We experimented with the real-world biological instances provided by DasGupta et al. [3]. We need similar amounts of time, but can solve them optimally. Moreover, we also experimented with synthetic data and further real-world instances to chart the border of feasibility of our algorithm.

2 Data Reduction Scheme

A data reduction rule reduces in polynomial time an instance to a smaller instance, without destroying the possibility of finding an optimal solution. Data reduction has proven useful as a general technique in coping with NP-hard problems [9]. The corresponding reduction rules, however, have to be developed in a problem-specific way. In this section, we describe an effective data reduction scheme for the Balanced Subgraph problem. The scheme seems to be applicable to a larger set of problems and deserves further investigation.

The idea is to find a small separator S (that is, a set of vertices whose deletion separates the graph into at least two components) that cuts off a small component C from the rest of the graph. Then, we replace S and C by a smaller gadget that exhibits the same behavior. The behavior is examined by exhaustively enumerating possible states of the separator and finding exact solutions to the small component C. A similar method has been suggested by Polzin and Vahdati Daneshmand [19] for the Steiner Tree problem. However, they do not employ gadgets and have no formal characterization of reducible cases.

Reduction Scheme. Let S be a separator and let C be a small component obtained by deleting S from the given graph G. Then, determine for each of the (up to symmetry) $2^{|S|-1}$ colorings of S the size of an optimal solution for the induced subgraph $G[S \cup C]$ and replace in G the subgraph $G[S \cup C]$ by a gadget that contains the vertices of S and possibly some new vertices.

The above scheme leaves open some details. Before filling them in, let us show a simple example.

Example 1. In Fig. 1, the separator S cuts off the vertices in C from the rest of the graph. Up to symmetry, there are only two possibilities how the vertices in S can be colored: equal or unequal. If they are colored equal (a), the subgraph $G[S \cup C]$ is balanced without edge deletions. Otherwise (b), one edge deletion is required. We can simulate this behavior with a single =-edge: it also incurs a cost of 0 when the two vertices of S are equally colored, and a cost of 1 otherwise. Therefore, we can replace the subgraph $G[S \cup C]$ by the gadget shown on the right.

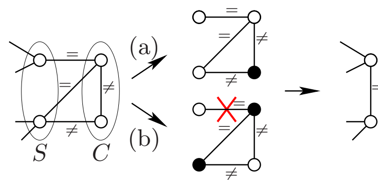


Fig. 1. Example for the data reduction scheme

To fully describe the reduction scheme, four questions have to be answered:

(a) The instances $G[S \cup C]$ have some vertices (those of the separator) pre-colored. How to solve these already partly colored instances?
(b) There is a combinatorial explosion with the sizes of S and C affecting the running time. Therefore, how do we restrict the choices of S and C?
(c) How can we efficiently find useful S/C-combinations?
(d) If existing, how can we construct a gadget that is smaller than $G[S \cup C]$ and correctly "simulates" $G[S \cup C]$?

Regarding (a), we reduce the instance to an instance without pre-colored vertices, and then solving the instance recursively. For this, we merge all vertices pre-colored black into a single uncolored vertex and all vertices pre-colored white into a single uncolored vertex. We then add $m := |E|$ edges labeled $\neq$ between these two vertices. Any solution for this instance will then color the two vertices differently, and we can (possibly by flipping all colors) reconstruct a solution for the pre-colored instance.

Regarding (b), this can be simply done by imposing a fixed limit. In our implementation, we restrict the size of S to at most 4, mainly because of difficulties with the gadget construction. The size of C is restricted by 32 (however, due to the structure of our instances, this limit did not play a role, because all components found were much smaller).

As to (c) and (d), we will answer these questions in the next two subsections.

2.1 Efficiently Finding Separators

As mentioned before, we effectively only search for separators of size at most 4. Having found such a separator, we check for the (hopefully) small component that is cut off in this way.

To improve running time, we special-case the search for separators of size 0 (that is, the graph consists of more than one connected component) and separators of size 1 (that is, articulation points). They can be found in linear time using depth-first search [7]. For these cases, the gadget construction can be omitted: the 2-connected components[1] can be treated independently, and optimal colorings of two components can always be merged (possibly by flipping all colors in one component), since they overlap only in one vertex. Note that this phase in particular removes all degree-1 vertices. Separators of size 2 can also be found in linear time [13]. However, we did not implement this algorithm, since it is quite complicated and error-prone to implement (several errors in the original publication have been pointed out [10]). Separators of size k for small k can be found efficiently by flow techniques [12]. However, after some experiments we settled for the subsequently described heuristic instead, which is faster and produces no worse results in our tests. Let $N(X) := \{u \mid \{u, v\} \in E \wedge v \in X\} \setminus X$. For each vertex v, set $C := \{v\}$ and iteratively enlarge C by a vertex v' that minimizes

[1] A set of vertices is 2-connected if there are at least two vertex-disjoint paths between any pair of vertices from this set.

the size of $S := N(C \cup \{v'\})$ until $|C| \geq 32$. Record all combinations of S and C with $S \leq 4$ found in this way.

To fix the order in which we try to reduce for an S/C-combination, we sort them primarily by increasing size of S and secondarily by decreasing size of C. In this way, one deals with the tentatively best data reduction candidates first. In our experiments, the finding of separators in the above way altogether never took more than few seconds.

2.2 Gadget Construction

The goal is to show how the subgraph $G[S \cup C]$ induced by the separator S and the small component C can be replaced by a smaller, "equivalent" subgraph (gadget). A simple case has already been described in Example 1. Now, we describe a general methodology, leading also to theoretically interesting problems that deserve further investigation. For lack of space, we defer the proofs to the full version of this paper.

Let us call a separator of size i simply *i-cut*. As mentioned before, it it easy to deal with 0- and 1-cuts. Hence, we focus on larger separators, thereby describing constructions delivering optimal gadgets in case of 2- and 3-cuts and a heuristic approach for 4-cuts. We also briefly discuss the mathematical and algorithmic challenge behind constructing gadgets for i-cuts for general i.

By an *optimal* gadget we refer to one with a minimum number of vertices. When speaking of an *equivalent* gadget which replaces the subgraph $G[S \cup C]$, we refer to a subgraph H with the following properties: Gadget H contains all vertices from S and possibly more; in particular, S forms the "interface" where H is plugged in instead of $G[S \cup C]$. Further, the original graph G has a solution for Balanced Subgraph of size k iff the modified graph where H replaces $G[S \cup C]$ has a solution of size $k' \leq k$, where the difference between k' and k is determined by the gadget. In particular, an optimal solution for G can be directly reconstructed from an optimal solution for the modified graph.

Gadget construction for 2-cuts. 2-cuts generalize Example 1. Up to symmetry, there are only two colorings of the two separator vertices called u and v. In each of these two cases, we compute recursively an optimal solution for $G[S \cup C]$, which can be done quickly, since only small S are considered.

Let n_e be the size of an optimal solution for $G[S \cup C]$ where u and v have the same color and let n_d be the size of an optimal solution where they have distinct colors. We perform the following gadget construction, where the gadget consists solely of vertices from S. If $n_e \geq n_d$, then remove C and edges within S and add $n_e - n_d$ edges labeled $\neq$ between u and v. Otherwise, remove C and edges within S and add $n_d - n_e$ edges labeled $=$ between u and v. Note that reducing 2-cuts in particular gets rid of all vertices of degree 2.

Lemma 1. *The described gadget replacement yields an equivalent instance.*

Gadget construction for 3-cuts. The basic approach is the same as for 2-cuts. The gadget construction, however, becomes more intricate. The idea is to

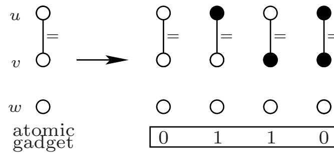


Fig. 2. Example for an atomic gadget with its cost vector

construct the final gadget from *atomic gadgets*, which can be added independently until in total they have the desired effect. To characterize the effect of an atomic gadget, we introduce the concept of a *cost vector*. In the case of 3-cuts, up to symmetry, we have four possibilities to color the separator vertices from S. For each case, we compute the cost of an optimal Balanced Subgraph solution of $G[S \cup C]$. For a fixed order of the colorings, these values build the cost vector of the form (c_1, c_2, c_3, c_4). The goal is then to find atomic gadgets such that their corresponding atomic cost vectors add up to the cost vector associated with $G[S \cup C]$.

We show that it is sufficient to consider atomic gadgets that, besides S, have at most one additional vertex. The first type of atomic gadgets are gadgets exclusively made of vertices from S. More specifically, there are six possibilities to put exactly one edge, either labeled $=$ or $\neq$, between the three possible vertex pairings in S. Each of these possibilities yields an atomic gadget. Moreover, each of these atomic gadgets naturally one-to-one corresponds to a cost vector with 0/1-entries. For instance, let $\{u, v, w\}$ form the separator. Then, the atomic gadget with a $=$-edge between u and v corresponds to the cost vector $(0, 1, 1, 0)$ (see Fig. 2): If u and v have the same color (once white, once black), then the insertion of the $=$-edge does not cause an inconsistency. Thus, we have an additional solution cost of 0, justifying the two zero-entries in the cost vector. If u and v have different colors, then the insertion of the $=$-edge causes an inconsistency, generating an additional solution cost of 1, justifying the two one-entries in the cost vector. Generalizing this to the five other possibilities of putting one labeled edge, we thus arrive at the following:

Lemma 2. *By inserting exactly one edge labeled $=$ or $\neq$ between the vertices from S, we obtain the six atomic cost vectors* $(0, 0, 1, 1)$, $(0, 1, 0, 1)$, $(0, 1, 1, 0)$, $(1, 0, 0, 1)$, $(1, 0, 1, 0)$, *and* $(1, 1, 0, 0)$.

All cost vectors in Lemma 2 have even parity. Hence, we need a second type of gadgets to be able to construct cost vectors with odd parity: gadgets that contain all vertices from S plus a new vertex connected to all vertices from S. We derive four atomic gadgets of this kind with different cost vectors, namely the cases that the edges connecting S to the new vertex are labeled $(\neq, \neq, \neq)$, $(=, =, \neq)$, $(\neq, =, \neq)$, or $(=, \neq, \neq)$.

Lemma 3. *By inserting one new vertex and connecting it to all vertices from S and assigning various edge labels, we obtain four atomic gadgets corresponding to the atomic cost vectors* $(0, 1, 1, 1)$, $(1, 0, 1, 1)$, $(1, 1, 0, 1)$, *and* $(1, 1, 1, 0)$.

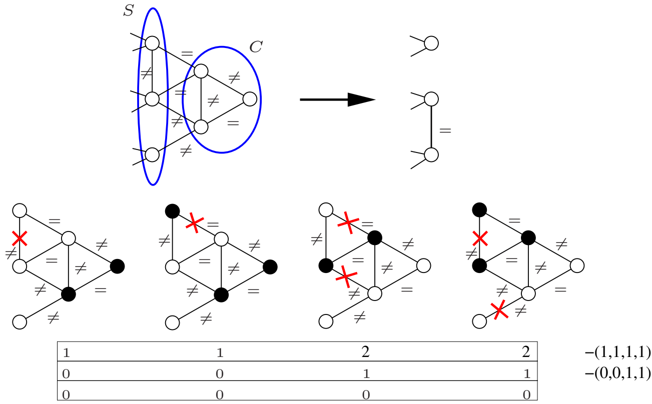


Fig. 3. Example for the construction of a gadget with $|S| = 3$

The four atomic cost vectors from Lemma 3 all have odd parity. In this sense, we now may speak of *even* or *odd* cost vectors.

Now, we can describe the general gadget construction. To do so, first note that vectors where all entries have the same value are easy because this means that the solution for $G[S \cup C]$ is independent of the coloring of S and hence one can simply remove C and all edges between vertices of S. Even further, this means that if we are given a cost vector (c_1, c_2, c_3, c_4), then without loss of generality we can *normalize* it by simply subtracting or adding the vector $(1, 1, 1, 1)$. Now, given a cost vector (c_1, c_2, c_3, c_4), the gadget construction task one-to-one corresponds to finding a way to subtract atomic cost vectors from (c_1, c_2, c_3, c_4) such that one receives the vector $(0, 0, 0, 0)$. If we arrive at a cost vector with at least two 0-entries that cannot be transformed into $(0, 0, 0, 0)$, then due to the above reasoning we may also add the vector $(1, 1, 1, 1)$. Altogether, this results in the following algorithm:

1. Compute the cost vector for given S and C.
2. Normalize the cost vector by subtracting the vector $(1, 1, 1, 1)$ until at least one entry becomes 0.
3. If the cost vector has odd parity and has more than one 0-entry, then add $(1, 1, 1, 1)$.
4. If the cost vector has odd parity, then subtract a suitable odd atomic cost vector (that is, one that does not produce negative entries).
5. While the vector is not $(0, 0, 0, 0)$, repeat:
   (a) If the cost vector has three 0-entries, then add $(1, 1, 1, 1)$.
   (b) Subtract a suitable even atomic cost vector that decreases the maximum entry.

We show an example in Fig. 3. On top we show the induced subgraph $G[S \cup C]$ and the gadget by which it can be replaced. In the middle we show optimal solutions for the (up to symmetry) four possible colorings of S and mark by a

cross the edges that have to be deleted. Cost vectors are displayed below these figures. The cost vector associated with $G[S \cup C]$ is shown in the top box. We then normalize by subtracting (1,1,1,1) and then subtracting the atomic cost vector (0,0,1,1) that corresponds to the gadget given on top, ending up with the zero-vector.

Theorem 1. *The above algorithm produces a gadget with the minimum number of vertices for every pair* (S, C) *where* S *is a 3-cut.*

Note that the construction is not necessarily optimal with respect to the number of edges introduced, nor with respect to the decrease in k. However, in our experiments these objectives rarely had different optimal solutions.

As a consequence of the considerations so far, we obtain the following result illustrating the power of our approach.

Corollary 1. *With the described data reduction scheme, all separators with* $|S| = 2$ *and* $|C| \geq 1$ *and and all separators with* $|S| = 3$ *and* $|C| \geq 2$ *are subject to data reduction.*

Gadget Construction for 4-Cuts and Outlook. The gadget construction for 3-cuts already has required quite some machinery. The case of 4-cuts becomes still much more involved due to the increased combinatorial complexity. A provably optimal gadget construction as for 3-cuts currently does not seem practically feasible. Thus, we have chosen a heuristic approach for finding and constructing gadgets for 4-cuts.

We conjecture that atomic gadgets with at most two vertices in addition to the four separator vertices suffice. Thus, we generated 2^6 atomic gadgets with no extra vertex, 2^4 atomic gadgets with one extra vertex, and 2^9 atomic gadgets with two extra vertices. We then filtered out those that can be obtained by combining cheaper ones, and arrived after about five minutes of computation time at a set of 2948 atomic gadgets. They are stored in a fixed lookup table.

Once given this toolbox of atomic gadgets, we again try to derive the all-zero vector in a way analogous to the case of 3-cuts. This procedure is now realized by an exhaustive branch&bound algorithm. We start with the normalized vector. Should this fail, the vector $(1, 1, 1, 1)$ is added once and the procedure is repeated. Each gadget vector is associated with a cost corresponding to its number of extra vertices; this number is minimized. In fact, it is not too hard to see that this algorithm produces for 3-cuts, when given the 10 atomic cost vectors, the same result as the algorithm given for 3-cuts.

The branch&bound algorithm works quite well for cost vectors with small entries, but can become a bottleneck for vectors with high entries. We examine a simple heuristic to mitigate this in Sect. 4.

We close with a description of challenges for further research that arise in our work with cost vectors. For this, we describe the scenario in a more abstract way.

Given a set S of n vectors of length l with nonnegative integer components, let a "linear combination" be a sum of some vectors of S, where vectors can occur multiple times (equivalently, have a positive integer scalar factor). Let a

"basis" be a set S that allows to obtain any vector of length l with nonnegative integer components as a linear combination. (The terms are chosen in analogy to vector spaces, but because of the nonnegative integer restriction, we do not have a vector space here.) We face the following problems: How to recognize whether a vector set is a basis? Given a basis and a target vector t, how to find a linear combination that produces t? Given a large set of vectors, how can we find a smallest or minimal basis?

In our work, we actually have a small modification of this problem because as single vector with negative components also the vector $(-1, -1, \ldots, -1)$ is allowed. Also, the vectors come at different costs (number of new vertices), and we would like to find linear combinations of minimum cost.

3 Fixed-Parameter Tractability

While the data reduction rules presented in Sect. 2 can often much reduce the instance size, there will typically remain a "core" that cannot be further reduced. To solve the remaining instances exactly while getting a useful worst-case time bound, we use a fixed-parameter algorithm.

We can prove the fixed-parameter tractability with respect to the parameter k of Balanced Subgraph by giving a parameter-preserving reduction from Balanced Subgraph to its special case Edge Bipartization.

Theorem 2. *Given an m-edge graph with at most k edge deletions allowed,* Balanced Subgraph *can be solved in $O(2^k \cdot m^2)$ time.*

Proof. Edge Bipartization can be solved in $O(2^k m^2)$ time [8]. It is the special case of Balanced Subgraph where all edges are labeled $\neq$. The following simple replacement transforms a Balanced Subgraph instance into an equivalent instance for Edge Bipartization: Replace every $=$-edge $\{u, v\}$ by two edges $\{u, x_{uv}\}$ and $\{x_{uv}, v\}$, where x_{uv} is a new vertex. It is straightforward to show that the transformed instance has a solution of size k iff the original instance has a solution of size k, and that from a solution of the transformed instance we can easily reconstruct a solution of the original instance. The transformation is computable in linear time and at most doubles the size of the the new instance. Hence, we obtain the same asymptotic running time as for Edge Bipartization. □

Theorem 2 improves an $O(n^{2L} \cdot (nm)^3)$ time exact algorithm by DasGupta et al. [3, Remark 1], where L is the number of $\neq$-edges (since clearly $k \leq L$).

In our implementation, we do not use the reduction from Edge Bipartization, but directly modify the Edge Bipartization algorithm to work for Balanced Subgraph. Further, we employ a heuristic speedup trick similar to the one used for an iterative compression algorithm for Vertex Bipartization [14]. We inferred speedups with this trick up to a factor of about 10^{12}. We refer to the full version of this paper for details.

4 Experimental Results

We applied our data reduction combined with the improved iterative compression routine to gene-regulatory networks and randomly generated graphs.

Besides the data reduction rules described in Sect. 2, we additionally delete self loops and pairs of edges sharing the same end vertices if the edges have different types. These reductions can be seen as special cases of our data reduction scheme from Sect. 2 with $|C| = 0$ and $|S| = 1$ and 2, respectively. Furthermore, we only replace a small component by a gadget if this leads to an improvement; that is, either the number of vertices is reduced, or, in the case of an equal number of vertices, the number of edges is reduced.

Additionally, we tested a heuristic running time improvement to circumvent a problem with the data reduction based on 4-cuts: For some instances the running time drastically increased because we encountered a cost vector with entries having high values. This increased the number of possible linear combinations and therefore the running time. An example appeared when the algorithm processed the regulatory yeast network: it ran into the cost vector $(2, 8, 8, 0, 31, 39, 39, 31)$ and therefore the instance could not be solved within several hours (whereas it could be solved without 4-cut reduction within minutes). To take advantage of 4-cut reductions without wasting hours of running time through such (rarely occurring) cases, based on experimental findings we introduced a new cut-off parameter. More precisely, we stop the gadget construction if the sum of the entries of a cost vector is more than 25. We experimentally show in the next two sections that this cut-off value is sufficient for the considered networks.

As a further comparison point, we implemented an integer linear programming (ILP)-based approach (we omit the details). However, it was consistently outperformed by the iterative compression approach as soon as the heuristic speedup mentioned in Sect. 3 was employed.

All experiments were run on an AMD Athlon 64 3400+ machine with 2.4 GHz, 512 KB cache, and 1 GB main memory running under the Debian GNU/Linux 3.1 operating system. The program was compiled with the Objective Caml 3.08.3 compiler and the GNU gcc 3.3.5 compiler with options "-O3 -march=athlon". For the approximation algorithm by DasGupta et al. [3], we used MATLAB version 7.0.1.24704 (R14). Our source code is available as free software from `http://theinf1.informatik.uni-jena.de/bsg/`.

Biological Networks. We started our experimental investigations with gene regulatory networks up to the size of about 700 vertices and more than 7000 edges.

We begin with comparing our algorithm to the approximation algorithm of DasGupta et al. [3]. The authors considered the regulatory networks of yeast and human epidermal growth factor (EGFR). We additionally examine a macrophage network [17]. The results of both algorithms are given in Table 1. Apart from giving an optimal solution instead of an approximative one, we can also decrease the running time for the yeast and macrophage networks from about one hour to less than a minute. Note, however, that the running time of the approximation algorithm could probably be much improved by implementing it in a more

Table 1. Comparison of approximation [3] and our exact algorithm. Here, t denotes the running time in minutes. For the approximation algorithm, "$k \leq$" is the solution size, and "$k \geq$" is the lower bound gained from the approximation guarantee. The approximation algorithm was run with 500 randomizations.

			Approximation			Exact alg.	
Data set	n	m	$k \geq$	$k \leq$	t	k	t
EGFR	330	855	196	219	7	210	108
Yeast	690	1082	0	43	77	41	1
Macrophage	678	1582	218	383	44	374	1

Table 2. Size of the largest component remaining and overall running time t (including solution by iterative compression) when reducing only separators up to size c

	Yeast			EGFR			Macrophage		
c	n	m	t	n	m	t	n	m	t
0	690	1080	91 s	329	783	> 15 h	678	1582	> 1 day
1	321	709	77 s	290	727	> 15 h	535	1218	> 1 day
2	173	469	11 s	167	468	> 15 h	140	397	> 1 day
3	155	424	4 s	99	283	> 15 h	113	335	ca. 1 day
4	?	?	> 5 h	89	259	108 min	70	228	4.5 h
4r	144	405	5.6 s	89	260	97 min	70	228	18 s

efficiently executed language, or simply by doing fewer randomized trials at the cost of a possibly worse result. For the macrophage network, we could compute an optimal solution of size $k = 374$. This emphasizes the importance of our data reduction rules, since for such high solution sizes the iterative compression algorithm (Theorem 2) cannot be applied directly. Furthermore, here it is remarkable that the network breaks up into several smaller components of up to 70 vertices that have to be solved by iterative compression independently, whereas for the other two networks only one large component remains after data reduction. As a further comparison point, the ILP-based approach was not able to solve the three instances even after applying the data reduction.

To investigate the power of our data reduction rules for different sizes c of the separator S, we investigated stepwise for c the results for the yeast, EGFR, and macrophage networks. The results are given in Table 2, where setting c to $4r$ means that we use a cut-off of 25 for the sum of the entries of a cost vector in the case of cut sets of size 4.

We denote applying our data reduction to a separator of size i by *ci-reduction*. The yeast network can already be solved with improved iterative compression and 2-reduction. In contrast, the EGFR network cannot be solved within reasonable time without also using 3- and 4-reduction. Regarding the macrophage network, the use of 4-cuts reduces running time severely.

We now investigate $c4$-reduction with and without cut-off value. For all networks, we could achieve the best data reduction results by using $c4r$-reduction: As mentioned above, for the yeast network the "normal" $c4$-reduction does not return any results within 5 hours. In contrast to the other entries for which we aborted the experiments in Table 2, here the running time is caused by the data reduction itself and not to the iterative compression routine. Therefore, we cannot give the size of the reduced graph. Setting the cut-off parameter to 25, we obtained an instance that is more reduced than by applying $c3$-reduction alone. The reason that we still cannot achieve a better overall running time is the running time for the $c4r$-reduction itself. For the EGFR network, the size of the largest component does hardly change going from $c4$- to $c4r$-reduction, indicating that we do not lose much by the cut-off; in fact, we achieve a better overall running time for $c4r$. Applying $c4r$-reduction instead of $c4$-reduction to the macrophage network does not change the size of the remaining largest component, but decreases the running time from hours to seconds.

Note that we really need the combination of data reduction and the improvements of iterative compression to solve the instances.

We also considered four small regulatory networks obtained from the Panther pathways database, consisting of about 100 vertices and up to 200 edges. With $c3$-reduction we could compute optimal solutions ranging from 20 to 28 in split seconds.

To end the section of regulatory networks, we describe our results for two larger networks that yet cannot be solved optimally with our method. Regarding the regulatory network for a toll-like receptor [18], we could reduce the number of vertices from 688 to 244 and the number of edges from 2208 to 1159 within three minutes. For the regulatory network of the archaeon *Methanosarcina barkeri* [5], we were less successful. The number of vertices was decreased from 628 to 500 and the number of edges from 7302 to 6845 in 25 minutes. This could be a hint that the very dense structure of this network is hard to attack by our data reduction.

Random Networks. To further substantiate our experimental results, we generated a test bed of random graphs with the algorithm described by Volz [21]. Thereby, we tried to model the yeast network by choosing the following settings: the cluster coefficient is set to 0.016, the distribution of node-degrees is set to power-law with $\alpha = -2.2$, and the probability to assign $\neq$ to an edge is set to 0.205.

We generated 5 instances each for graph sizes ranging from 100 to 1000 vertices. The number of edges of the generated instances is slightly more than 1.5 times of the number of vertices. We investigated the power of our data reduction by computing the number of vertices and edges of the reduced instances. Table 3 shows the average results for instances of each size. Independent of the instance size, about 75% of the vertices are reduced. Note that this is also true for the yeast network that we try to model.

The results given in Table 3 are received with setting the cut-off parameter for the $c4$-rule again to 25. Redoing the test with a higher threshold of 50 did in

Table 3. Reduction effect for random networks. Average over 5 instances for each column. Here, n is the number of vertices in the original graph, n' is the number of vertices after data reduction, m' is the number of edges after data reduction, and t is the running time in seconds.

n	100	200	300	400	500	600	700	800	900	1000
m	172.6	336.8	492.4	640.2	791.2	970.6	1108.8	1286.6	1435.6	1585.6
n'	29	48.8	75	95	119.8	153.2	169.2	193.4	211.6	239.6
m'	102.3	165.8	252	324	398.4	518	565.8	672.4	734.6	815.8
t	1	7	6	5.5	6	8.5	8	15.5	18.5	15.5

no case change the number of reduced edges or vertices by more than one, but increased the running time for some instances from seconds to several hours.

Considering the size of instances that can be solved optimally by improved iterative compression after data reduction, here the threshold seems to be at graphs with about 500 vertices. Three out of the five instances could be optimally solved in up to 20 hours, where the sizes of the optimal solutions are between $k = 76$ and $k = 91$. Note that the solution sizes are higher than for the yeast network, which has more than 600 vertices and an optimal solution of size 41. Because of this, the random instances seem to be somewhat more difficult than the yeast network itself, which is consistent with observations by DasGupta et al. [3].

5 Outlook

There are numerous avenues for future research. DasGupta et al. [3] also introduced a directed version of the Balanced Subgraph problem. The approximation results are worse than for the undirected case, which is probably why there is no implementation yet [3]. Fortunately, the directed case can be reduced to the Vertex Bipartization problem, which can be solved in $O(3^k \cdot mn)$ time [14]. Again, this opens the route for experimental studies.

In principle, our data reduction scheme is applicable to all graph problems where a coloring of the vertices is sought. This includes problems where a subset of the vertices is sought, such as Vertex Cover or Dominating Set. However, it remains to find appropriate gadgets constructions for problems other than Balanced Subgraph. Further, it would be nice to have a formal characterization of graphs for which our separation-based data reduction scheme is useful.

Acknowledgement. We thank the authors of [3] for making their source code available. Further, we thank our students Tamara Steijger and Thomas Zichner for help with the experiments and Jiří Matoušek (Prague) for helpful references for the integer linear combination problem.

References

1. Avidor, A., Langberg, M.: The multi-multiway cut problem. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 273–284. Springer, Heidelberg (2004)
2. Chiang, C., Kahng, A.B., Sinha, S., Xu, X., Zelikovsky, A.Z.: Fast and efficient bright-field AAPSM conflict detection and correction. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26(1), 115–126 (2007)
3. DasGupta, B., Enciso, G.A., Sontag, E.D., Zhang, Y.: Algorithmic and complexity results for decompositions of biological networks into monotone subsystems. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 253–264. Springer, Heidelberg (2006)
4. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
5. Feist, A.M., Scholten, J.C.M., Palsson, B.Ø., Brockman, F.J., Ideker, T.: Modeling methanogenesis with a genome scale metabolic reconstruction of Methanosarcina barkeri. Molecular Systems Biology, 2(2006.0004) (2006)
6. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006)
7. Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Information Processing Letters 74(3–4), 107–114 (2000)
8. Guo, J., Gramm, J., Hüffner, F., Niedermeier, R., Wernicke, S.: Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. Journal of Computer and System Sciences 72(8), 1386–1396 (2006)
9. Guo, J., Niedermeier, R.: Invitation to data reduction and problem kernelization. ACM SIGACT News 38(1), 31–45 (2007)
10. Gutwenger, C., Mutzel, P.: A linear time implementation of SPQR-trees. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 77–90. Springer, Heidelberg (2001)
11. Harary, F.: On the notion of balance of a signed graph. Michigan Mathematical Journal 2(2), 143–146 (1953)
12. Henzinger, M.R., Rao, S., Gabow, H.N.: Computing vertex connectivity: New bounds from old techniques. Journal of Algorithms 43(2), 222–250 (2000)
13. Hopcroft, J.E., Tarjan, R.E.: Dividing a graph into triconnected components. SIAM Journal on Computing 2(3), 135–158 (1973)
14. Hüffner, F.: Algorithm engineering for optimal graph bipartization. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 240–252. Springer, Heidelberg (2005)
15. Khot, S.: On the power of unique 2-prover 1-round games. In: Proc. 34th STOC, pp. 767–775. ACM Press, New York (2002)
16. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press, Oxford (2006)
17. Oda, K., Kimura, T., Matsuoka, Y., Funahashi, A., Muramatsu, M., Kitano, H.: Molecular interaction map of a macrophage. Research Report, http://www.systems-biology.org/001/010.html (August 2004)
18. Oda, K., Kitano, H.: A comprehensive, map of the toll-like receptor signaling network. Molecular Systems Biology, 2(2006.0015) (2006)
19. Polzin, T., Vahdati Daneshmand, S.: Practical partitioning-based methods for the Steiner problem. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 241–252. Springer, Heidelberg (2006)
20. Vazirani, V.V.: Approximation Algorithms. Springer, Heidelberg (2001)
21. Volz, E.: Random networks with tunable degree distribution and clustering. Physical Review E, 70:056115 (2004)
22. Zaslavsky, T.: Bibliography of signed and gain graphs. Electronic Journal of Combinatorics, DS8 (1998)

Algorithms for the Balanced Edge Partitioning Problem

Ekaterina Smorodkina, Mayur Thakur, and Daniel Tauritz

Department of Computer Science,
University of Missouri-Rolla, Rolla, MO USA
{eas7d3,thakurk,tauritzd}@umr.edu

Abstract. We consider the problem of minimizing communication overhead while balancing load across cooperative agents. In the past, similar problems have been modeled as the balanced node partitioning problem, where the objective is to partition the nodes into components such that each component has roughly the same number of nodes while the number of edges connecting components is minimized. We describe some real-world scenarios where one needs to find partitions in which all components have an approximately equal number of edges, while minimizing the number of edges connecting components. We introduce the (k, r)-Balanced Edge Partitioning problem to model this type of scenario and present approximation algorithms for this problem on certain graphs. In addition, we present five heuristics for the restricted case of the problem. We evaluate these heuristics on three kinds of graphs: power network-like graphs, preferential attachment graphs, and the class of *spatial preferential attachment graphs* that we introduce in this paper. Our results show that the choice of the heuristic with the best results depends on the properties of the input graph and the quality of our solution depends on the initial conditions.

Keywords: Graph partitioning, balanced graph partitioning, heuristics, Kernighan-Lin heuristic.

1 Introduction

In many multiagent settings, k agents cooperatively solve a problem that is subdivided into a set of k subproblems and each agent is assigned one subproblem. Such situations arise, for example, when cooperative agents solve constraint satisfaction problems [6]. Other examples of cooperative distributed problem solving occur in distributed control systems of electric power grids, manufacturing systems, and sensor networks. In this paper, we introduce a novel graph-theoretic problem that models optimal work distribution among agents on some problems. In the (k, r)-Balanced Edge Partitioning ((k, r)-BEP) problem, where $r \geq 1$ and $k > 1$, we are given a graph $G = (V, E)$ and asked to partition the vertices into k components such that the number of edges in each component is no more than $r\frac{|E|}{k}$ and the number of edges between components is minimized.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 311–323, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The problem of partitioning a graph into components with equal numbers of vertices while minimizing the number of edges between the components, and its applications in parallel computing, have been thoroughly studied [1,10]. However, we are unaware of any literature about partitioning graphs to minimize the number of edges between components such that the number of *edges* in each component is equal. We study the latter problem and its application to optimal load balancing for cooperative agent-based problem solving.

The contributions of this paper are as follows: First, we show how an example problem can be modeled as a (k, r)-BEP problem. Second, we show that the (k, r)-BEP problem is not only NP-hard, but has no polynomial time approximation algorithm with finite approximation factor when $r = 1$ and k is part of the input (unless P = NP). Third, we present approximation algorithms for special classes of graphs – planar graphs and degree-bounded graphs. Fourth, we describe several heuristics for the problem of $k = 2$ (that is, in the two-agent case); these heuristics can later be extended for $k > 2$. We test our heuristics on instances of real-world electric power network-like graphs. Fifth, we construct a novel preferential attachment-like generative model for spatial networks so that our algorithms can be tested on many instances of different sizes. Generative models of networks are useful because large real-world instances are hard to obtain.

1.1 Motivation

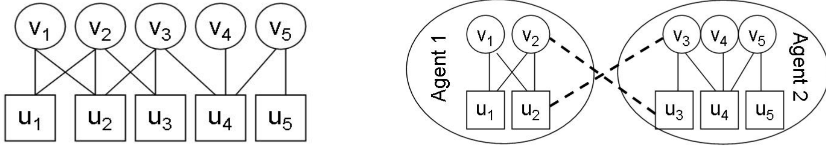
Consider solving a system of N non-linear equations with N unknowns:

$$\mathbf{f}(\overline{x}) = 0, \tag{1}$$

where $\mathbf{f}$ is a vector of N functions and $\overline{x}$ is a vector of N variables. The Newton-Raphson method is a well known approach for solving this problem. At each iteration $k + 1$, this method finds a more accurate solution, $\overline{x}^{k+1}$, as follows: It computes the Jacobian of the system of equations evaluated at the current solution, $[\mathbf{J}(\overline{x}^k)]$, solves the linear system

$$[\mathbf{J}(\overline{x}^k)][\overline{d}] = -[\mathbf{f}(\overline{x}^k)], \tag{2}$$

and computes $\overline{x}^{k+1} = \overline{d} + \overline{x}^k$. This process continues until $||\mathbf{f}(\overline{x}^k)|| < \epsilon$, where ϵ is some predefined error tolerance. If the Jacobian matrix is sparse, the speed of finding $\overline{d}$ can be significantly improved by using a "smart" matrix reordering method [3]. The number of operations needed to compute $\overline{d}$ depends on the number of non-zero entries in the Jacobian matrix – the more non-zero entries there are, the more operations are needed to solve for $\overline{d}$. We can construct a graph $G = (V, E)$ from the original system of equations 1 as follows: For each equation f_i, create a vertex $u_i \in V$ and for each variable x_j, create a vertex $v_j \in V$. For any variable x_j occurring in an equation f_i, create an undirected edge $(u_i, v_j) \in E$. Then the number of non-zero entries in the Jacobian matrix equals the number of edges in G.



(a) Graph G constructed from the system of equations (3). (b) Optimal partitioning of G for cooperative 2-agent problem solving.

Fig. 1. Example of partitioning a graph constructed from the system of equations (3)

An example of forming a graph from a system of equations is now presented. Consider the following system of equations:

$$\mathbf{f}(\overline{x}) = \begin{cases} f_1 : x_1 x_2 + x_1 = 0 & f_2 : 2x_1 + x_2^2 + \frac{1}{2}x_3 = 0 \\ f_3 : x_2 x_3 + 5 = 0 & f_4 : x_3^2 - 5x_4 x_5 = 0 \\ f_5 : x_5 + 3 = 0. & \end{cases} \quad (3)$$

The system of equations (3) contains five equations with five unknowns. The graph constructed from this system is shown in Figure 1(a).

Consider k cooperative agents solving a system of equations in parallel. In this scenario, the variables are partitioned into k sets, X_1, X_2, ..., X_k, and the equations are partitioned into k sets, F_1, F_2, ..., F_k. Each agent a_i controls one set of variables X_i to solve one set of equations F_i. If the graph constructed from the system of equations has k disconnected components of roughly equal sizes, then the variables and the equations can be partitioned into sets of roughly equal sizes in such a way that each agent can solve its set of equations autonomously. In general, however, if variables and equations are distributed among the agents, communication between the agents will be required. To solve a set of equations F_i, agent a_i will need to obtain information about the variables that occur in F_i but are not controlled by a_i (let us call such variables external variables for a_i). It can then use the Newton-Raphson method to solve its set of equations and obtain the updated information about external variables for a_i. Each agent communicates values it obtains for an external variable x to the agent that controls x. Suppose there is a mechanism in place that will drive this process to convergence.[1] Also, suppose that convergence criteria have been defined.

Under these assumptions, the optimal partitioning of variables among the agents is equivalent to partitioning the vertices of G into k components so that the number of edges between components is minimized (thus minimizing the number of variables shared between the agents) and the number of edges within each component is about the same (thus ensuring that work is equally distributed among the agents). The optimal partitioning of the graph for our example system of equations (3) is shown in Figure 1(b).

A real-world application of the (k, r)-BEP problem is partitioning an electric power network with the power flow along all lines in each partition controlled

[1] Such mechanisms are beyond the scope of this paper.

by an agent. The calculations of the power flow through a network (or a section of a network) depend on the number of lines present.

1.2 Notation and Definitions

The following symbols and definitions will be used throughout this paper. For each finite set X, $|X|$ denotes the cardinality of X. $G = (V, E)$ denotes an undirected graph where V is a set of vertices and E is a set of edges.

We first define the standard graph partitioning problem where the objective is to create components with roughly equal numbers of vertices while minimizing the number of edges connecting components. For each positive integer k and each $\epsilon > 0$, the (k, ϵ)-Balanced Node Partitioning ((k, ϵ)-BNP) problem [1] is defined as follows:

Definition 1. *Given a graph, $G = (V, E)$, partition the vertices of G into k components, $V_1, V_2, ..., V_k$, so that for all $i = 1, 2, ..., k$, $|V_i| \leq (1 + \epsilon)\frac{|V|}{k}$ and the number of edges in the set $\{(u, v) \in E : u \in V_i, v \in V_j, i \neq j\}$ is minimized.*

We now formally define the main computational problem introduced and studied in this paper. For each $k > 1$ and a value $r \geq 1$, we define the (k, r)-BEP problem as follows:

Definition 2. *Given a graph, $G = (V, E)$, partition the vertices of G into k components, $V_1, V_2, ..., V_k$, so that for all $i = 1, 2, ..., k$, $|E_i| \leq r\frac{|E|}{k}$ (where $E_i = \{(u, v) \in E \,\big|\, u \in V_i, v \in V_i\}$) and the number of edges in the set $\{(u, v) \in E : u \in V_i, v \in V_j, i \neq j\}$ is minimized.*

At first glance, one might think that in the strict version of the latter problem ($r = 1$) the number of edges in each partition is *exactly* $|E|/k$. However, this is not necessarily the case because there will in general be edges connecting partitions. Note that in both (k, ϵ)-BNP and (k, r)-BEP problems, the parameters k, ϵ (and k, r respectively) are not part of the input. We will use Balanced Node Partitioning (BNP) and Balanced Edge Partitioning (BEP) to denote versions of these problems when the parameters are part of the input.

1.3 Related Work on Graph Partitioning

The definition of the (k, r)-BEP problem is very similar to that of the (k, ϵ)-BNP problem. There is a significant amount of literature on the latter problem, while the former one has not been given much attention. Because of the similarities between the two problems, it is possible that the methods used to approximate the solutions to the instances of the (k, ϵ)-BNP problem could be applicable to the (k, r)-BEP problem. That is why we present a brief overview of the existing literature on the (k, ϵ)-BNP problem in this section.

One of the most popular heuristics for the (k, ϵ)-BNP problem is that of Kernighan and Lin [7]. In their approach, the input graph is split into k partitions and vertices are greedily swapped between pairs of partitions until a local

optimum is found. This heuristic is known to perform well in practice. A detailed description of this and various other heuristics for the balanced graph partitioning problem can be found in [5].

In [9] Saran and Vazirani gave an approximation algorithm for the $(k, 0)$-BNP problem with the approximation guarantee of $1 - 1/k$. This algorithm runs in polynomial time for a fixed k; otherwise in exponential time (exponential in the value of k). Simon and Teng [10] used an idea similar to that of the Kernighan-Lin heuristic to give an approximation algorithm for $(k, 1)$-BNP with an approximation guarantee of $O(\log(|V|)\log(k))$. Finally, in [1], Andreev and Räke gave an approximation algorithm for the (k, ϵ)-BNP problem with an approximation guarantee of $O(\log^2 |V|)$ for any fixed $\epsilon > 0$.

Whether or not the (k, r)-BEP problem can be nontrivially approximated is open for fixed $k \geq 2$ and fixed $r \geq 1$. In this paper we examine several heuristics that can be used on the restricted case of the (k, r)-BEP problem (specifically when $k = 2$ and $r = 1$), as well as classes of graphs for which the approximation algorithms to the (k, ϵ)-BNP problem are also approximation algorithms to the (k, r)-BEP problem.

2 Algorithms and Heuristics

2.1 Hardness of the BEP Problem

It is known that for $\epsilon = 0$, the BNP problem has no polynomial time approximation algorithm with finite approximation factor unless P = NP [1]. To establish the hardness of the BEP problem we use a reduction similar to the one in [1] and show that the BEP problem has no polynomial time approximation algorithm with finite approximation factor unless P = NP.

Theorem 1. *The BEP problem has no polynomial time approximation algorithm with finite approximation factor unless* P = NP.

Proof. The 3-Partition problem is defined as: Given a set of $n = 3m$ integers, $a_1, a_2, ..., a_n$, and an integer A, such that $\frac{A}{4} < a_i < \frac{A}{2}$ and $\sum_{i=1}^{n} a_i = mA$, decide whether the given integers can be partitioned into sets of three such that each set of three adds up to A. We know that a 3-Partition problem is strongly NP-complete [4]. That is, it remains NP-complete even if the numbers a_i and A are polynomially bounded.

Given an instance $X = \langle a_1, a_2, \ldots, a_n, A\rangle$ of a 3-Partition problem, we construct a graph G such that for each number a_i, G contains a subgraph $G_{a_i} = (V_{a_i}, E_{a_i})$, where $V_{a_i} = \{v_0^{a_i}, \ldots, v_{a_i}^{a_i}\}$ and $E_{a_i} = \{(v_0^{a_i}, v_1^{a_i}), \ldots, (v_{a_i-1}^{a_i}, v_{a_i}^{a_i})\}$. So $G = (V, E)$, where $V = \bigcup_{i=1}^{n} V_{a_i}$ and $E = \bigcup_{i=1}^{n} E_{a_i}$. (Note that this construction can be achieved in polynomial time if the a_i's and A are polynomial in the size of the instance.) Then the following holds: X is a yes instance of the 3-Partition problem if and only if the value of an optimal solution to G (looked upon as an instance of the BEP problem with $k = m$ and $r = 1$) is 0.

Therefore, if for some $f \geq 1$, there is an f-approximation algorithm to the BEP problem, then it will be able to solve the 3-Partition problem where the

numbers in the input are polynomial-sized. This contradicts the assumption that P $\neq$ NP. □

Note that since no approximation algorithm exists for BEP, good heuristics are our best hope. There may exist approximation algorithms for restricted cases of (k, r)-BEP or even of BEP.

2.2 Approximation Algorithms for Planar and Degree Bounded Graphs

We present approximation algorithms to the (k, r)-BEP problem for planar and degree bounded graphs. Many real-world network graphs fit this category, so these algorithms can be useful in practice. Our approximation algorithms work via a reduction to an instance of the (k, ϵ)-BNP problem for which we know an approximation algorithm with an approximation guarantee of $O(\log^2 |V|)$, where $|V|$ is the number of vertices in the graph [1]. If OPT_V is the cost of the optimal solution of the reduced instance of the (k, ϵ)-BNP problem, then our algorithm for the (k, r)-BEP problem finds a cut no bigger than $O(\log^2 |V|)OPT_V$, where $|V|$ is the number of vertices in the graph.

If $|V|$ is the number of vertices in a *planar* graph, then this graph can have no more than $3|V| - 6$ edges [8]. We use this fact to prove that an approximation algorithm for (k, ϵ)-BNP can be used to approximate solutions for (k, r)-BEP.

Theorem 2. *Given a planar graph $G = (V, E)$ with average vertex degree d_a, we can find a solution to the (k, r)-BEP of G for all $r > 6/d_a$ with a cost of at most $O(\log^2 |V|)OPT_V$. Here OPT_V is the cost of the optimal solution to (k, ϵ)-BNP of G with $\epsilon = d_a r/6 - 1$.*

Proof. Let $\epsilon = \frac{d_a r}{6} - 1$. Then $\epsilon > 0$, since $r > 6/d_a$. We can now find a (k, ϵ)-BNP of G within $O(\log^2 n)$ of OPT_V, using the approximation algorithm from [1]; let $V_1, V_2, ..., V_k$ be the partitions found. Then for all $i = 1, 2, ..., k$, $|V_i| \leq (1+\epsilon)\frac{|V|}{k}$. Let $E_i = \{(u, v) \mid u \in V_i, v \in V_i\}$. Since G is a planar graph, we know that any subgraph of G with at most $|V_i|$ nodes has at most $3|V_i| - 6$ edges. Therefore, for all $i = 1, 2, ..., k$, $|E_i| \leq 3|V_i| - 6 \leq 3(1+\epsilon)\frac{|V|}{k} - 6$. Since the average vertex degree of G is d_a, we know that $|V| = 2|E|/d_a$. So $|E_i| \leq 3(1+\epsilon)\frac{2|E|}{kd_a} - 6$. Substituting $\frac{d_a r}{6} - 1$ for ϵ we get $|E_i| \leq 3(1 + \frac{d_a r}{6} - 1)\frac{2|E|}{kd_a} - 6 \leq r\frac{|E|}{k}$. Therefore, the solution to the (k, ϵ)-BNP of G satisfies the constraints of the (k, r)-BEP of G and its cost is at most $O(\log^2 n)OPT_V$. □

Theorem 2 is useful when the average vertex degree of a graph is greater than 3. If the average vertex degree of a graph is between 2 and 3 and $k = 2$, then Theorem 2 relaxes the problem too much, allowing any partition to contain any number of edges. A similar analysis can be applied to graphs with bounded vertex degree.

Theorem 3. *Given a graph $G = (V, E)$ such that the degree of all vertices in G is bounded by a constant d, and average vertex degree is d_a, we can find a*

solution to the (k, r)-BEP of G with a cost of at most $O(\log^2 |V|)OPT_V$, for all $r > d/d_a$. Here OPT_V is the cost of the optimal solution to the (k, ϵ)-BNP of G with $\epsilon = rd_a/d - 1$.

Proof. Let $\epsilon = \frac{rd_a}{d} - 1$. Then $\epsilon > 0$, because $r > d/d_a$. We can now find (k, ϵ)-BNP of G within $O(\log^2 n)$ of OPT_V, using the approximation algorithm from [1]; let $V_1, V_2, ..., V_k$ be the partitions found. Then for all $i = 1, 2, ..., k$, $|V_i| \leq (1+\epsilon)\frac{|V|}{k}$. Let $E_i = \{(u, v) \,|\, u \in V_i, v \in V_i\}$. Since the degree of all vertices of G is bounded by d, we know that any subgraph of G with $|V_i|$ nodes has at most $d|V_i|/2$ edges. Therefore, for all $i = 1, 2, ..., k$, $|E_i| \leq d|V_i|/2 \leq d(1+\epsilon)\frac{|V|}{2k}$. Since the average vertex degree of G is d_a, we know that $|V| = 2|E|/d_a$. So $|E_i| \leq d(1+\epsilon)\frac{2|E|}{2kd_a} = d(1+\epsilon)\frac{|E|}{kd_a}$. Substituting $\frac{rd_a}{d} - 1$ for ϵ we get $|E_i| \leq d(1 + \frac{rd_a}{d} - 1)\frac{|E|}{kd_a} = r\frac{|E|}{k}$. Therefore, the solution to the (k, ϵ)-BNP of G satisfies the constraints of the (k, r)-BEP of G and its cost is at most $O(\log^2 n)OPT_V$. □

2.3 Heuristics for the (2, 1)-BEP Problem

We now provide five heuristics for the restricted case of the (k, r)-Balanced Edge Partitioning problem when $k = 2$ and $r = 1$.

Ratio heuristic first greedily creates a feasible solution and then greedily improves it until a local optimum is reached. Given a graph and two vertex partitions it moves vertices from the larger to the smaller partition one at a time, each time moving the vertex that would ensure that the number of edges in the smaller partition is no more than the maximum allowed. Furthermore, the vertex to be moved is selected such that moving it to the smaller partition maximizes the ratio of the number of inner edges in the smaller partition to the size of the cut. This is repeated until a feasible solution is achieved. The produced cut is improved by greedly moving one vertex at a time from one partition to another such that the solution remains feasible. The initial partitions are created by simply placing the highest degree vertex into one partition and the rest of the vertices into the other one.

Breadth First Search Ratio (BFS-Ratio) heuristic first identifies multiple sets of vertices that should be placed into the same partition and constructs a vertex weighed graph. Each identified set of vertices becomes one vertex in the new graph, whose weight equals to the number of edges between the vertices in the set. The edges connecting the selected sets of vertices become the edges of the new graph. The weighted version of the Ratio heuristic is applied to the new graph. In the weighted version of the Ratio heuristic, the number of edges in a partition equals to the sum of vertex weights plus the number of edges between the vertices of that partition.

The sets of vertices of the original graph that should be placed into the same partition are identified by the breadth first search (BFS) as follows. Start from the node with the highest degree and iteratively perform BFS. At step i add all nodes that are a distance i from the root as long the number of edges added at

step i is at least a factor e of the number of edges present at the end of step $i-1$ (in our experiments e was set to the average vertex degree). When this condition fails, place all the discovered vertices into one partition, remove them from the original graph and perform BFS again. Continue until all vertices are discovered.

Kernighan-Lin heuristic [7] can be used to minimize the cut between two partitions of a graph with equal numbers of vertices. On input $G = (V, E)$ this heuristic initializes two partitions V_1 and V_2 such that $|V_1| = |V_2|$ and then greedily identifies sets $X \subset V_1$ and $Y \subset V_2$, where $|X| = |Y|$ such that swapping X and Y would locally minimize the cut between the two partitions. The final cut produced by this heuristic largely depends on the initial partitions. The solution produced by this heuristic is not necesseraly a feasible solution for the $(2, 1)$-BEP problem; if this is the case the solution can be turned into a feasible one using the same method as in the Ratio heuristic. We refer to this heuristic as the Kernighan-Lin Ratio (KL-Ratio) heuristic.

Edge Kernighan-Lin (EKL) heuristic: A modification of the Kernighan-Lin heuristic can be made to produce feasible solutions for the $(2, 1)$-BEP problem. This heuristic initializes two partitions V_1 and V_2 to a feasible solution of the $(2, 1)$-BEP problem. Then it identifies sets $X \subseteq V_1$ and $Y \subseteq V_2$ such that swapping X and Y would locally minimize the cut between the two partitions and result in a feasible solution to the $(2, 1)$-BEP problem. This is done as in the original Kernighan-Lin heuristic but without the requirement that $|X| = |Y|$. In the original Kernighan-Lin heuristic at each step of the Optimization Algorithm described in [7], one vertex from V_1 is added to X and one vertex from V_2 is added to Y. Three cases are possible in our modification of the heuristic: (1) a vertex from V_1 is added to X and a vertex from V_2 is added to Y, (2) a vertex from V_1 is added to X and nothing is added to Y, and (3) a vertex from V_2 is added to Y and nothing is added to X.

We expect that in our modification of the Kernighan-Lin heuristic the initial partitions have a big influence on the final solutions produced by the heuristic. We used two methods to generate initial partitions: (1) creating a feasible solution as it is done in the Ratio heuristic and (2) using the locality principal: start with the highest degree vertex in one partition and continue adding the vertices closest to it until a feasible solution is achieved. We refer to these initialization methods followed by modified Kernighan-Lin heuristic as EKL-Ratio and EKL-Local respectively.

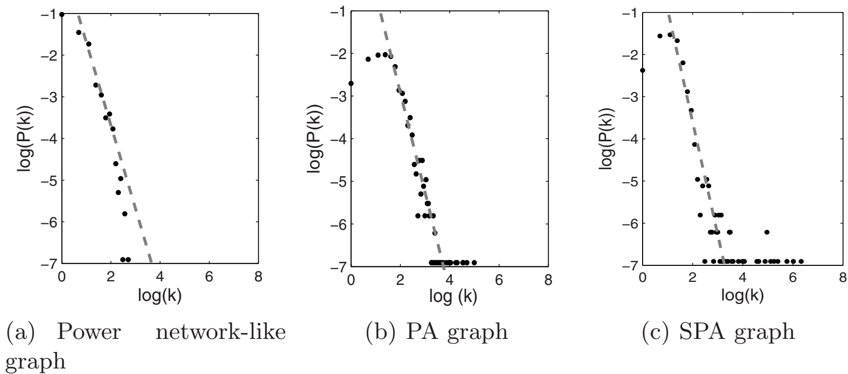
3 Experimental Results

The only known way of finding the exact optimal solutions to instances of the $(2, 1)$-BEP problem takes exponential time and therefore we cannot compute the optimal value, even for modest-sized instances. To assess our heuristics we experimentally evaluated them and compared the results.

Graphs arising from real-world networks are most interesting for practical purposes. Because large real-world graphs are not easily obtainable, test graphs were generated with real-world graph models.

3.1 Graph Models

Three graph models were used to generate test cases: (a) the Preferential Attachment (PA) model as described in [2], (b) the Spatial Preferential Attachment (SPA) model (a new graph model that emerged from this research), and (c) subgraphs of a large real-world power network. Our SPA model was motivated by the spatial property of real-world networks as well as by the power-law scaling of the PA model. To generate a graph, the SPA model starts with a small clique of vertices (just as in the PA model). At each iteration a new vertex is added and it randomly chooses one of the existing vertices as its "location." Next, the new vertex preferentially attaches to m other vertices that lie within a radius r of its location vertex, where m and r are predefined parameters. The probability of a newly added vertex attaching to another vertex, v, within the radius r, is proportional to the vertex degree of v. Thus, vertices attach preferentially, just as in the PA model, but the set from which a new vertex chooses its neighbors is not the set of existing vertices, but the subset of vertices that lie within a radius r of the location vertex. Note that a detailed study of the SPA model might be of independent interest and is not part of this work.



(a) Power network-like graph

(b) PA graph

(c) SPA graph

Fig. 2. Degree distribution of the three graph types used in the experiments

The number of vertices in the graphs generated with PA and SPA models ranged from 500 to 1000, with the value of parameter m ranging from 2 to 5; the value of r was set to 3. Each of these two graph models generated a total of 84 test graphs. Proprietary data of a large real-world power network was used to generate 30 power network-like graphs with 1000 nodes each; this was done by randomly selecting a root node and performing breadth first search until 1000 nodes were found.

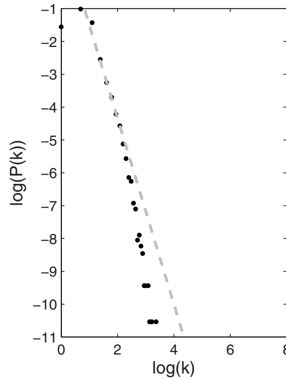


Fig. 3. Degree distribution of the large power network used to generate power network-like graphs

The degree distribution of a graph depends on its type, defined by the model used to generate it. Examples of degree distributions of the three graph types used in the experiments are shown in Figure 2, where k is the degree and $P(k)$ is the probability that a vertex has degree k. This figure indicates that our power network-like graphs did not have the power law scaling property, while the graphs generated with the PA and SPA models did. To verify that our power network-like graphs have characteristics of the real-world power networks, the degree distribution of the large real-world power network that was used to generate our power network-like graphs is shown in Figure 3. Figure 3 reveals the lack of power-law scaling in the real-world power network, the characteristic that is also observed in the power network-like graphs, Figure 2(a), used in the experiments.

3.2 Results

The five heuristics described in this paper were evaluated on the graphs described in Section 3.1. The Chaco 2.0 software package [5] was used to find equally sized (in the number of vertices) partitions of the graphs needed for the KL-Ratio heuristic. This software package uses the multilevel Kernighan-Lin heuristic [5] to find such partitions while minimizing the cut.

Table 1. Ranking of the heuristics on a scale from 1 to 5 (1 being the best, i.e. found the smallest cuts, and 5 being the worst) for each graph type used in the experiments

Graph Type	1	2	3	4	5
Power network-like graphs	KL-Ratio	BFS-Ratio	EKL-local	EKL-Ratio	Ratio
PA graphs	EKL-Ratio	EKL-Local	Ratio	KL-Ratio	BFS-Ratio
SPA graphs	EKL-Local	EKL-Ratio	Ratio	KL-Ratio	BFS-Ratio

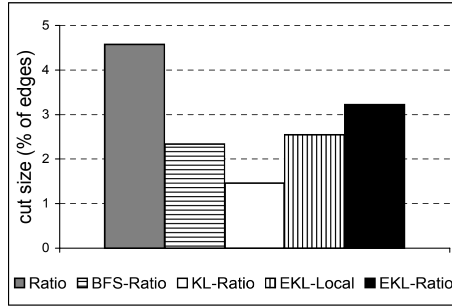
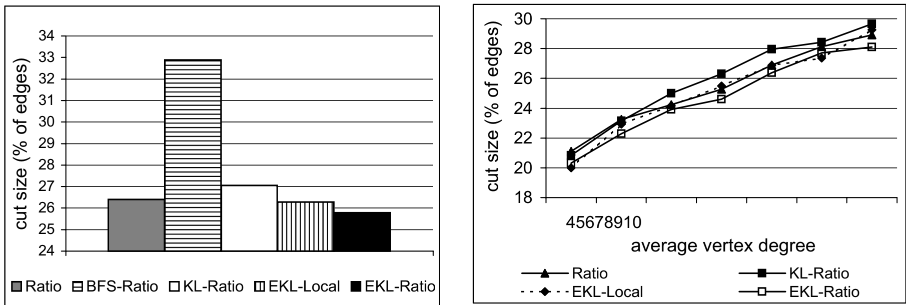


Fig. 4. Average percentage of edges in cuts found by each heuristic on power network-like graphs with 1000 nodes

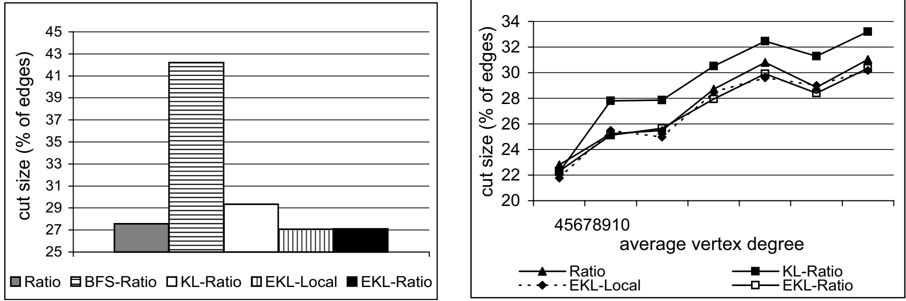


(a) Average percentage of edges in cuts found by each heuristic.

(b) Average percentage of edges in cuts found by each heuristic versus average vertex degree.

Fig. 5. Heuristics evaluated on PA graphs

The size of the cut was measured as percentage of the number of edges in the graph. The results are presented in Figures 4 – 6. These results show that the performance of the heuristics depends on the graph type. In the power network-like graphs, the KL-Ratio heuristic found the smallest cuts. The next smallest cuts were obtained by the BFS-Ratio heuristic followed by EKL-Local, EKL-Ratio, and finally the Ratio heuristic. The performance of these heuristics differs drastically when evaluated on PA and SPA graphs. On PA graphs EKL-Ratio on average finds the smallest cuts. The heuristic that was in second place on the power network-like graphs, BFS-Ratio, finds the largest cuts on PA graphs. A similar situation is observed on SPA graphs, where the smallest cuts are found by EKL-Local and the largest cuts are produced by the BFS-Ratio heuristic. The heuristic that was in the first place on power network-like graphs is only fourth best (out of five) on the other two graph types. Table 1 shows the order (from best to worst) of the heuristics for each graph type, computed based on the size of the cut produced by the heuristics. For PA and SPA graphs, where power law scaling was observed, the order of the heuristics is similar, unlike that



(a) Average percentage of edges in cuts found by each heuristic.

(b) Average percentage of edges in cuts found by each heuristic versus average vertex degree.

Fig. 6. Heuristics evaluated on SPA graphs

of power network-like graphs, where power-law scaling was absent. This suggests that the presence of power law scaling in a graph influences which heuristic (out of the proposed five) will find the smallest cut on that graph.

Figure 5(b) and Figure 6(b) show the relationship between the size of the cut produced by the four best heuristics versus the average vertex degree for PA and SPA graphs. These figures indicate that we can expect larger cuts as the average vertex degree increases.

4 Summary and Conclusions

In this paper we have introduced the (k, r)-BEP problem and its application to cooperative agent-based computing. We presented approximation algorithms for special cases of the problem, when input graphs are planar or degree bounded. We developed five heuristics for the constrained version of this problem, two of which use a modification of the well-known Kernighan-Lin heuristic. These heuristics were experimentally evaluated on graphs generated with real-world graph models as well as on power network-like graphs. The results verify the hypothesis that the performance of our modification of the Kernighan-Lin heuristic depends on the initial partitions. Furthermore, our results show that the choice of the best heuristic (out of the proposed five) depends on the type of graph used as the problem instance.

Acknowledgments. We thank Mariesa Crow for allowing us to use the proprietary data and we thank anonymous referees for helpful comments.

References

1. Andreev, K., Räcke, H.: Balanced graph partitioning. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on parallelism in algorithms and architectures, pp. 120–124. ACM Press, New York, NY, USA (2004)

2. Barabási, A., Albert, R.: Emergence of scaling in random networks. Science 286, 509–512 (1999)
3. Davis, T.: A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. ACM Transactions on Mathematical Software 30(2), 165–195 (2004)
4. Garey, M., Johnson, D.: Computers and Intractability. In: Garey, M., Johnson, D. (eds.) A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, New York (1979)
5. Hendrickson, B., Leland, R.: The Chaco user's guide version 2.0. Technical Report SAND95-2344, Sandia National Laboratories, Albuquerque, NM (July 1995)
6. Jung, H., Tambe, M.: Performance models for large scale multiagent systems: using distributed POMDP building blocks. In: AAMAS '03. Proceedings of the second international joint conference on autonomous agents and multiagent systems, pp. 297–304. ACM Press, New York, NY, USA (2003)
7. Kernighan, B., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell Systems Journal 49, 291–307 (1972)
8. Rosen, K.: Discrete Mathematics and Its Applications, 5th edn. McGraw Hill, New York (2003)
9. Saran, H., Vazirani, V.: Finding k-cuts within twice the optimal. SIAM Journal of Computing 24(1), 101–108 (1995)
10. Simon, H., Teng, S.: How good is recursive bisection? SIAM Journal of Scientific Computing 18(5), 1436–1445 (1997)

Experimental Evaluations of Algorithms for IP Table Minimization

Angelo Fanelli, Michele Flammini, Domenico Mango,
Giovanna Melideo, and Luca Moscardelli

Dipartimento di Informatica, Università di L'Aquila,
Via Vetoio Loc. Coppito, 67010 L'aquila, Italy
{angelo.fanelli,flammini,mango,melideo,moscardelli}@di.univaq.it

Abstract. The continuous growth of the routing tables sizes in backbone routers is one of the most compelling scaling problems affecting the Internet and has originated considerable research in the design of compacting techniques. Various algorithms have been proposed in the literature both for a single and for multiple tables, also with the possibility of performing address reassignments [1,5].

In this paper we first present two new heuristics, the BFM and its evolution called BFM-Cluster, that exploit address reassignments for the minimization of $n > 1$ routing tables, and their performances are experimentally evaluated together with the already existing techniques. Since a main problem posed by the growth of the routing tables sizes is the consequent general increase of the table lookup time during the routing of the IP packets, the aim is twofold: (i) to measure and compare the compression ratios of the different techniques and (ii) to estimate the effects of the compression on the lookup times by measuring the induced improvement on the time of the main algorithms and data structures for the fast IP address lookup from the original tables to the compressed ones. Our point is that the existing methods are efficient in different situations, with BFM-Cluster heuristic outperforming all other ones.

Keywords: Routing, IP protocol, compression, lookup times, optimal and approximation algorithms.

1 Introduction and Motivations

In the Internet Protocol (IP) communications between hosts are possible by means of interconnected forwarding elements called routers. Each router consists of input interfaces, output interfaces, a forwarding engine and a routing table. Exchanged messages are arranged into packets or datagrams. Unlike the circuit-switched networks, every packet travels across the network independently of all others. This implies that each packet is labelled with both a globally unique source and a destination address, which it must carry along. In a router, the bottleneck in packet forwarding is to lookup the destination address of an incoming

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 324–337, 2007.
© Springer-Verlag Berlin Heidelberg 2007

packet in the routing database, that is to determine the output interface through which the packet must be forwarded.

Unfortunately, the huge and disorganized growth of the Internet during the last years has caused an excessive increase in the number of entries of the routing tables. Beside their memory requirements, the main problem posed by this phenomena is the consequent general increase of the table lookup time during the routing of IP packets. Thus, a considerable research effort has been devoted in the design of techniques for reducing the size of IP routing tables.

In [5] the authors presented an optimal polynomial time algorithm (*Optimal Routing Table Constructor* - ORTC) for constructing a routing table that has the least possible number of entries, while still providing the same routing information. Moreover, they experimentally evaluted ORTC by showing that it reduces the number of prefixes by around 40%.

The envisaged close enhancement of the IP protocol to version 6 urgently requires the solution of the IP routing tables minimization problem in a new and more effective way, that is by performing address reassignments. Such a facility can actually be exploited also inside the current version of the protocol, thanks to the introduction of the so called *Network Address Translators*, NATs for short, by which independent address reassignments are possible inside subnetworks [6]. In this scenario, efficient tables minimization algorithms exploiting addresses reassignments are of crucial importance. Motivated by the above discussion and by exploiting address reassignments, in [1] the authors provided a new polynomial time optimal algorithm, called BF, that minimizes the size of single routing tables and a $3h$-approximation algorithm (h is the length of the IP addresses), called BF-Multi, that minimizes the sum of the sizes of $n > 1$ routing tables.

Starting from the above results, in this paper we first introduce two new heuristics, the BFM heuristic and its evolution called BFM-Cluster, that exploit address reassignments for the minimization of $n > 1$ routing tables, and then we focus on the experimental evaluation of the above mentioned techniques for IP tables minimization (i.e., ORTC, BF, BF-Multi, BFM and BFM-Cluster). Since a main problem posed by the growth of the routing tables sizes is the consequent general increase of the table lookup time during the routing of the IP packets, the aim is twofold: (i) to measure and compare the compression ratios of these techniques and (ii) to estimate the effects of the compression on the lookup times by measuring the induced improvement of the time of the main algorithms and data structures for the IP address lookup (e.g., Binary Search on prefix Length [18,19], Multi-ary Tries [15,16] and LC-Trie [14]).

The paper is organized as follows. In the next section we present the stat-of-art in IP address tables minimization; in Section 3 we introduce our new heuristics. Section 4 is devoted to the implementation details such as software, hardware, requirements and metrics. Section 5 illustrates our contributions on the comparison of the compression ratios of the table minimization algorithms and on the effect of tables minimization on the IP lookup data structures. Section 6 concludes by analyzing experimental results and discussing directions for

future work. We apologize for the many omissions and missing details, but space constraints imposed several limitations.

2 Preliminaries

In the IP protocol each host is assigned a unique address of $h = 32$ bits, with h presumably higher in future versions. Routers consist of input interfaces, output interfaces, a forwarding engine and a routing table. Each routing table is a list of entries, where each entry e consists of three different fields: a network mask $mask_e$, a destination network address $dest_e$ and a next-hop address $next_e$. Field $mask_e$ consists of a h bit binary string of the form $1^{l_e}0^{h-l_e}$, where l_e is called the length of the mask. The network address $dest_e$ is a h bit binary string representing the IP address associated to the entry, and the next-hop address $next_e$ identifies the index of the output interface corresponding to the entry.

A given IP address matches entry e if its leading l_e bits are coincident with the respective l_e ones of $dest_e$. Consider two entries e and f such that $l_e < l_f$ and the leading l_e bits of $dest_f$ exactly matching the ones of $dest_e$. It is easy to see that any IP address matching entry f matches also entry e. We then say that there is an *inclusion* of entry f in entry e.

Since an IP address can match different entries, routing is performed on a longest mask matching entry base, that is the output interface chosen to forward a packet is taken from the matching entry having the longest mask. The efficient solution of such a problem, called *Longest Matching Prefix*, has given rise to different algorithms, some of which are briefly presented in the following subsections, together with an overview of the minimization ones implemented for the experimentation. Due to space limitations their presentation is necessarily incomplete, with many details just mentioned and/or left unspecified. However, the interested reader can refer to the given literature for a more detailed description.

2.1 IP Lookup Algorithms

Many fast route lookup algorithms have been proposed in the last few years [2,3,4,8,9,11,14,16,19]. Based on the data structure used, these algorithms can be classified into one of the following three categories: trie-based, comparison-based and hash-based. The trie-based algorithms use the traditional key search idea and organize the routing table into a tree-like data structure [7]. Each node in the trie has zero or more child nodes. Each lookup of a key starts at the root of the trie and then walks down to find the longest match. The idea underlying comparison-based algorithms is mainly related to the binary search scheme. Modifications have been devised to accommodate the prefix into sorted arrays. In hash-based algorithms, the results of hashing functions are used as indexes into memory. The perfect hash function completes the lookups in only one memory access that can achieve the highest searching speed.

The particular algorithms used in this paper to evaluate the effect of table minimization on their lookup times are among the ones with the best performances:

Binary Search on prefix Length [18,19], which combines comparison-based and hash-based techniques, and Multi-ary Tries [15,16] and Level Compressed (LC) Trie [14], both belonging to trie-based category.

2.2 Table Minimization Without Address Reassignment

The Optimal Routing Table Constructor (ORTC) is an algorithm given in [5] which reduces the number of entries in a routing table.

Given a routing table that provides forwarding information for IP addresses using longest prefix matching, ORTC produces a new routing table that has the *same forwarding behavior* and *the least possible number of entries*.

A binary tree representation is used to graphically depict a set of prefixes. Each successive bit in a prefix corresponds to a link to a child node in the tree, with a 0 corresponding to the left child and a 1 corresponding to the right child. Note that the binary tree generally contains more nodes than prefixes, since every successive bit in the prefix produces a node. The nodes are labeled with next-hop information, typically a small integer or a set of small integers. Roughly speaking, ORTC optimizes a routing table using three steps over the binary tree representation. The first step propagates routing information down to the trees leaves. The second step finds the most prevalent next hops, by propagating information (sets of next hops) from the leaves back towards the root. In fact, shorter prefixes close to the root of the tree should route to the most popular or prevalent next hops. Finally, a third step moves down the tree, choosing a next hop from the set of possibilities for a prefix and eliminating redundant routes.

The space and time complexity of ORTC algorithm are $O(hN)$, where N is the number of entries in the input routing table.

2.3 Table Minimization with Address Reassignment

In [1] the authors propose algorithms for the minimization of routing tables using address reassignments. This approach differs substantially from the ORTC's one, where the hosts must maintain their original addresses, and allows to improve the effect of minimization by assigning IP addresses so as to obtain the maximum possible compression.

An optimal polynomial time algorithm (called BF) was presented in [1] for the case of single routing tables.

Let us briefly describe the underlying idea.

Denoted as a_i the number of hosts reached through the i-th output interface, if inclusions between entries are not allowed, then the set of the addresses matching any entry e in the table has cardinality 2^{h-l_e} and consequently the minimum number of entries corresponding to the i-th output interface is at least equal to the minimum number of powers of 2 whose sum is equal a_i. This clearly corresponds to the number of bits equal to 1 in the $h+1$ bit binary string that encodes a_i. Let $one(a_i)$ denote such a number. Then the minimum size of the table is at least $\sum_{i=1}^{\delta} one(a_i)$, where δ is the number of output interfaces. Moreover, it is not difficult to show that such a number of entries is always achievable.

Allowing the inclusion of one entry f in one entry e with $next_e \neq next_f$ and $l_e < l_f$, modifies the number of addresses matching the entries corresponding to the output interface $next_e$ from a_{next_e} to $a_{next_e} + 2^{h-l_f}$. As a consequence, the number of entries corresponding to the output interface $next_e$ becomes $one(a_{next_e} + 2^{h-l_f})$. Concerning the effect of such an inclusion on the entries of the output interface $next_f$, it is possible to charge a cost of one to the inclusion to keep track of the fact that such an entry will be effectively realized inside e, while the number of addresses matching the remaining entries of $next_f$ becomes $a_{next_f} - 2^{h-l_f}$. Exploiting such ideas, algorithm BF constructs a table of minimum size in time $O(h\delta)$.

Unfortunately, in the general case in which we are interested in minimizing the sum of the sizes of $n > 1$ tables[1], as shown in [1] this problem is *NP-hard*, but there exists a $3h$-approximation algorithm, called BF-Multi, that exploits a matrix representation of the instances of the problem. In fact, the routing behavior of any router r_j can be represented by means of a boolean matrix A^j in which each row is associated to a destination host and each column to one output interface of r_j. Let A be the *global matrix* given by the horizontal concatenation of all the matrices A^j. Let us define a *segment* in a given column of A as a maximal vertical sequence of consecutive entries equal to 1 in the column. The approximation algorithm is based on the idea that any permutation π of the rows of A corresponds in a natural way to an assignment of addresses to the hosts. Namely, the host corresponding to row i after the permutation receives the IP address given by the h bit binary string encoding $i - 1$. Since each segment corresponds to a limited number of entries in the IP routing table, a permutation causing a low number of segments in A yields also a solution with a low overall number of table entries. One of such permutations is than determined by reducing the problem to *minimum metrical TSP*.

3 The New Heuristics

In this section we introduce new heuristics for the problem of minimizing the sum of the sizes of $n > 1$ tables with address reassignment.

We emphasize that in both heuristics we focus on the address reassignment, i.e. we care about assigning addresses to host such that the total size of the routing tables can be minimized. On the other hand, we discard optimization issues relative to the minimization of each of the final routing tables obtained after the address reassignment; this is due to the fact that, after the addresses have been reassigned, such a problem is equivalent to the one of minimizing a routing table without address reassignment, which is optimally solved by the *ORTC* algorithm.

All the details of both heuristics will be shown in the full version of the paper.

[1] Notice that without address reassignment the problem can be trivially solved by independently applying ORTC to each single table.

3.1 BFM Heuristic

The main idea of the *BFM* heuristic is to construct a new virtual router $\bar{r}$ starting from the n routers as follows. For each host x we compute a n-tuple $\mathcal{U}_x = (out_x^1, \ldots, out_x^n)$ containing the n next hop interfaces associated to the host in each router, i.e. out_x^i is the next hop interface associated to x in router r_i. Let $\mathcal{U}$ be the set of all the obtained tuples. For each $u \in \mathcal{U}$, let H_u be the set of hosts corresponding to the n-tuple u. We run the BF algorithm on the new virtual router $\bar{r}$, in which each n-tuple u represents a virtual output interface with $|H_u|$ associated hosts. Finally, starting from the address assignment determined by BF, we construct each of the n final tables by selecting the corresponding next hop interfaces from the virtual next hops, i.e. from the n-tuples.

3.2 BFM-Cluster(k) Heuristic

This heuristic is an evolution of the BFM one just described and as an input parameter k, whose tuning is discussed in subsection 5.2. Since the size of the tables compressed by BF is usually very close to the number of output interfaces, in the BFM-Cluster heuristic we try to reduce the number of virtual output interfaces by clustering the tuples with a low number of not coinciding coordinates. In particular, when constructing the virtual router $\bar{r}$, we partition the set of tuples in clusters $c_1, c_2, \ldots$ such that tuples with a low fixed number of different components are in the same cluster, and finally, for each cluster c_i, we add to $\bar{r}$ a virtual output interface with a number of associated hosts equal to the sum of the numbers of hosts associated to each tuple in c_i. More specifically, the partition process is as described in the following. First of all, let us define tuples *d-close* if the number of their different components is at most d. We maintain an (initially empty) set $\bar{\mathcal{U}}$ of *leader* tuples, and analyze one by one all the tuples in $\mathcal{U}$: for each $u \in \mathcal{U}$, if u is k-close to a leader tuple $\bar{u} \in \bar{\mathcal{U}}$, we add u to the cluster whose leader is $\bar{u}$, otherwise we create a new cluster having u as leader. Finally, as in the BFM heuristic, we have to construct each of the n final tables. To this aim we first partition the set of addresses assigned to each cluster c_i between the tuples belonging to c_i, and then we proceed by selecting from the tuples the next hop interfaces relative to each table.

4 Methods Testing

We implemented the techniques described in the Section 2 in the C language. Overall, the developed C code consists of about 3000 lines. Since we are interested in the compression ratio and in the relative improvement of the lookup time, we performed the experimentation on a personal computer with a 2-GHz Pentium 4 processor and 256 MB of RAM running Windows XP.

4.1 Test Data

The techniques are tested on 15 existing routing tables, downloaded from the routing table snapshots provided by the IPMA Project [13,17] and from the

route server lists provided in [10,12], as well as on 56 artificially generated tables by independently modifying the next hop of each entry of a starting original table with a fixed probability between 10% and 30%.

For the minimization techniques, we considered 13 input instances composed with existing routing tables, and 7 instances composed with artificially generated ones. More precisely, for every set of tables we have run the implemented compression techniques and then constructed the auxiliary data structures both for the initial and compressed tables. In order to test the improvement times of the IP address lookup algorithms, each technique has been tested against three different traffic files, each containing $15 \cdot 10^6$ IP addresses on the original tables and on the compressed ones. Overall, we performed about 3000 different tests. More precisely, one traffic file is obtained as a permutation and repetition of the IP addresses originating (i.e., matching at least one entry) from the considered routing tables, and the other two traffic files contain random IP addresses.

Table 1 describes the existing routing tables features (i.e., the number of the entries in the routing table, the number of distinct next-hops found in the table and the date of tables snapshots), and the sets of the tables[2] considered in our tests. In order to have a more compact and readable presentation, we omit the description of the artificially generated tables and the one of the sets of these tables.

Table 1. Existing routing tables and their sets used as input instances

Tables					Sets												
ID	Table	Date	# Entries	# Next hop	1	2	3	4	5	6	7	8	9	10	11	12	13
1	utah.rep.net	02-11-04	153	6		x			x		x	x	x				
2	mae-west	24-08-97	15050	57	x			x				x	x	x	x	x	x
3	aasd	24-08-97	20328	19	x							x	x				
4	pb	24-08-97	20637	3	x							x	x				
5	mae-east	24-08-97	38470	59	x			x				x	x	x	x	x	x
6	funet	19-11-97	41709	20	x							x	x	x	x	x	x
7	as5388	02-11-04	62531	112		x		x	x		x	x	x	x	x	x	x
8	wcg.net	02-11-04	121800	898		x			x		x	x	x	x	x	x	x
9	ip.att	02-11-04	145682	23		x			x	x	x	x	x				
10	he	02-11-04	140112	196		x			x	x	x	x	x				x
11	ip.tiscali	02-11-04	145648	1			x		x	x	x	x					
12	opentransit	02-11-04	153317	15			x	x	x	x	x	x	x		x	x	
13	gbls	02-11-04	154740	303			x	x		x	x	x	x	x	x	x	x
14	oregon-ix	02-11-04	161635	53			x			x	x	x	x			x	
15	colt.net	02-11-04	162008	1			x			x	x	x					

[2] Unfortunately, we haven't current snapshots for mae-east, aads, pb, funet and mae-west. Anyway, we refer to these tables of the 1997 because they were taken as input in the main experimentation works about IP address lookup and IP tables minimization [5,18].

4.2 Measurement Principles

The main metric used to evaluate the IP tables minimization algorithms is the compression or reduction ratio, defined as $\frac{d-d_c}{d}$, where d is the initial size of the table and d_c is the size after the compression. More precisely, we have based our experimental analysis on the comparison between the total number of the table entries produced by the algorithms and the initial one. Moreover, referring to the IP lookup algorithms implemented, we have evaluated in percent terms the improvement of the lookup time (defined as $\frac{t-t_c}{t}$, where t is the lookup time on the initial table and t_c is the lookup time on the compressed one) achieved starting from the compressed tables with respect to the one yielded by the original ones.

5 Experimental Results

In order to have a more compact and readable presentation giving a direct indication of our experimental outcome, we present the results in global way. Moreover, since the tests of the compression algorithms and of the IP address lookup techniques on the artificially generated tables lead to almost the same results and considerations, we describe only the experimental results concerning the real world tables. Finally, concerning the IP address lookup techniques, we show both the global results relative to all the three traffic files, and the ones relative only to the "first" traffic file, i.e., the one containing addresses obtained from the existing routing tables.

5.1 The Case of a Single IP Routing Table

Table 2 shows the compression ratios obtained by running both the ORTC and the BF algorithms on the original tables in the case of a single routing table.

Table 2. Compression ratios of the original tables in the case of a single routing table

	ORTC	BF
Compression ratio	57.66%	99.80%

We notice the excellent compression results provided by the BF algorithm which can in fact exploit the address reassignment. As an example, after the compression obtained by BF, the routing table "mae-east" presents only 133 entries versus 38470 entries of the original table. Concerning the ORTC algorithm, the results pointed out by the experimental study reflect the theoretical analysis and the experimental evaluation presented in [5].

The effects of the compression on the lookup times are shown in tables 3 and 4 where we provide the average reduction of the lookup times of the implemented algorithms.

Table 3. Lookup times improvement of the IP address lookup algorithms executed on all the traffic files with respect to the original times on uncompressed routing tables

	Multibit	Binary search	LC-Trie
ORTC	8.56%	12.31%	0.65%
BF	77.88%	64.90%	32.22%

Table 4. Lookup times improvement of the IP address lookup algorithms on the first traffic file with respect to the original times on uncompressed routing tables

	Multibit	Binary search	LC-Trie
ORTC	9.53%	1.59%	12.21%
BF	87.22%	69.60%	43.19%

5.2 The Case of Multiple IP Routing Tables

Tables 5, 6, 7 and 8 show the experimental results in the case of compression of multiple IP routing tables. More precisely, Table 5 shows the reduction ratios on the sets of routing tables described in Table 1, Table 6 groups such results by set cardinality, whereas table 7 and 8 provide a measure of the induced improvement of the time of the main algorithms and data structures for the fast IP address lookup from the original tables to the compressed ones. Moreover, tables 9, 10 and 11 show the experimental results of the IP lookup algorithms on the single sets of routing tables described in Table 1.

Table 5. Compression ratios in the case of multiple routing tables

Set	ORTC	BF-Multi	BF-Multi + ORTC	BFM	BFM + ORTC	BFM-C.(1)	BFM-C.(1) + ORTC
1	33.53%	84.30%	94.44%	94.19%	95.92%	80.52%	96.77%
2	48.86%	55.53%	83.77%	85.29%	90.11%	35.62%	92.11%
3	67.20%	95.04%	98.07%	97.85%	98.78%	89.24%	98.79%
4	55.55%	76.74%	91.82%	92.70%	94.52%	66.73%	95.28%
5	56.48%	56.46%	83.67%	82.46%	89.88%	33.22%	92.72%
6	63.16%	80.62%	93.01%	90.99%	94.97%	64.03%	96.00%
7	60.29%	48.42%	81.96%	74.14%	86.94%	24.69%	91.09%
8	57.66%	12.18%	69.24%	45.08%	75.97%	−39.10%	83.74%
9	51.29%	−12.97%	60.78%	38.86%	69.61%	−55.60%	79.32%
10	51.97%	50.60%	82.80%	81.23%	87.60%	16.79%	90.74%
11	53.72%	44.76%	80.28%	78.62%	86.26%	15.91%	90.17%
12	52.37%	25.01%	73.60%	68.55%	80.30%	−4.28%	86.27%
13	53.28%	43.10%	79.91%	78.24%	86.13%	15.42%	90.00%

We evaluated the performance of the BF-Multi, BFM and BFM-Cluster(k) algorithms also with an additional compression step obtained by running ORTC

Table 6. Compression ratios in the case of multiple routing tables, grouped by set cardinality

#tables	ORTC	BF-Multi	BF-Multi + ORTC	BFM	BFM + ORTC	BFM-C.(1)	BFM-C.(1) + ORTC
5	57.16%	79.66%	92.61%	93.10%	95.31%	69.35%	96.08%
6	51.97%	50.60%	82.80%	81.23%	87.60%	16.79%	90.64%
7	57.70%	60.18%	85.60%	83.92%	90.26%	37.35%	92.86%
8	52.37%	25.01%	73.60%	68.55%	80.30%	−4.28%	86.27%
10	60.29%	48.42%	81.96%	74.14%	86.94%	24.69%	91.09%
13	51.29%	−12.97%	60.78%	38.86%	69.61%	−55.60%	79.32%
15	57.66%	12.18%	69.24%	45.08%	75.97%	−39.10%	83.74%

Table 7. Average lookup times improvement of the IP address lookup algorithms executed on all the traffic files with respect to the original times on uncompressed routing tables

	Multibit	Binary search	LC-Trie
BF-Multi	46.68%	38.16%	29.54%
BF-Multi + ORTC	59.35%	49.06%	−4.34%
BFM	61.63%	51.34%	−19.93%
BFM + ORTC	62.34%	46.77%	8.40%
BFM-Cluster(1)	37.41%	−11.73%	−17.67%
BFM-Cluster(1) + ORTC	65.44%	51.98%	16.51%

on their output tables. In fact, while on one hand BF-Multi does not exploit inclusions of entries (optimized by ORTC), that is each IP address matches at most one entry, on the other hand the BFM and BFM-Cluster(k) heuristics do not guarantee the minimality of the output tables, since the same output interface of a table may be associated to many virtual output interfaces (n-tuples).

In order to tune the parameter k of BFM-Cluster(k), we have executed the heuristic for different values of k. Since the number of tables to be minimized simultaneously is never greater than 15, we have obtained better results for small values of k, and the best ones (presented in the tables) for $k = 1$.

6 Analysis of the Results and Future Work

First of all, we point out how the reduction in the lookup times is higher when executing the lookup algorithms on the traffic file containing addresses obtained from the considered routing tables (i.e. the first traffic file). Thus, it results that the lookup time for addresses not matching any entry of a table is poorly affected by the size of the routing table.

The experimentation shown the effectiveness of the BF algorithm for the compression of a single table. It is worth noticing that BF could obtain a higher

Table 8. Average lookup times improvement of the IP address lookup algorithms executed on the first traffic file with respect to the original times on uncompressed routing tables

	Multibit	Binary search	LC-Trie
BF-Multi	57.57%	48.13%	47.87%
BF-Multi + ORTC	65.94%	55.50%	28.19%
BFM	82.62%	65.64%	15.31%
BFM + ORTC	82.99%	59.50%	37.83%
BFM-Cluster(1)	66.69%	35.87%	7.27%
BFM-Cluster(1) + ORTC	83.66%	60.72%	39.06%

Table 9. Average lookup times improvement on the algorithms executed on all the traffic files with respect to the original times on uncompressed routing tables (BF-Multi algorithm)

	LC Trie		Multibit		Binary search on IP length	
Set	BF-Multi	BF-Multi+ORTC	BF-Multi	BF-Multi+ORTC	BF-Multi	BF-Multi+ORTC
1	21.80%	−3.90%	57.44%	36.86%	37.54%	40.67%
2	30.88%	−6.14%	42.86%	48.45%	32.68%	42.95%
3	55.13%	24.80%	68.45%	66.46%	72.29%	73.59%
4	36.90%	14.32%	53.19%	43.04%	50.85%	58.98%
5	28.01%	−9.19%	44.89%	46.39%	34.01%	46.51%
6	44.69%	10.08%	58.57%	55.59%	59.03%	66.54%
7	33.56%	−9.43%	47.38%	48.11%	36.41%	50.05%
8	23.50%	−7.24%	40.16%	32.56%	25.48%	42.00%
9	18.61%	−16.38%	29.45%	27.71%	14.14%	31.55%
10	22.72%	−16.19%	42.75%	32.83%	32.62%	47.03%
11	19.87%	−13.83%	39.67%	27.59%	35.12%	43.96%
12	23.18%	−11.87%	39.44%	29.67%	29.07%	47.41%
13	25.28%	−11.51%	42.72%	34.60%	36.87%	46.61%

compression thanks to the address reassignment facility, which was not allowed to ORTC. Concerning the IP lookup times, the multibit and binary search on IP length algorithms present a better performance on the compressed tables with respect to LC-Trie. As foreseen, the lookup times are lower when the tables are compressed by the BF algorithm, with times ranging from 32.22% to 77.88%, depending on the lookup algorithm.

For the case of multiple tables, we can observe two factors that negatively affect the performances: (i) the number of tables in the sets and (ii) the "*homogeneity*" of the tables. In fact, both the BF-Multi algorithm and the BFM and BFM-Cluster ones degrade when the number of tables in the sets increases (see Tables 6 and 12). However, we have shown that it is possible to obtain a good average compression by further running the ORTC algorithm on the arising compressed tables. For all the algorithms, we have noticed that the compression

Table 10. Average lookup times improvement on the algorithms executed on all the traffic files with respect to the original times on uncompressed routing tables (BFM algorithm)

	LC Trie		Multibit		Binary search on IP length	
Set	BFM	BFM+ORTC	BFM	BFM+ORTC	BFM	BFM+ORTC
1	10.49%	10.68%	41.07%	68.91%	67.86%	38.09%
2	−30.42%	−0.80%	57.50%	62.50%	59.31%	45.16%
3	31.41%	32.13%	67.11%	77.55%	74.70%	73.72%
4	31.25%	19.24%	73.39%	62.10%	69.80%	51.00%
5	−16.40%	8.16%	58.75%	62.59%	67.62%	46.23%
6	2.59%	27.18%	69.18%	73.63%	70.67%	68.80%
7	−28.52%	9.21%	61.12%	66.03%	64.45%	50.34%
8	−111.07%	−3.39%	0.80%	53.57%	48.69%	34.38%
9	−105.17%	−11.56%	2.68%	51.52%	47.68%	28.14%
10	−17.35%	5.84%	56.40%	59.61%	65.50%	44.80%
11	−42.02%	6.71%	41.51%	57.96%	41.66%	44.93%
12	−0.10%	0.82%	66.24%	55.73%	60.27%	40.51%
13	16.25%	4.94%	71.68%	58.67%	62.98%	41.85%

Table 11. Average lookup times improvement on the algorithms executed on all the traffic files with respect to the original times on uncompressed routing tables (BFM-Cluster(1) algorithm)

	LC Trie		Multibit		Binary search on IP length	
Set	BFM-C.(1)	BFM-C.(1) + ORTC	BFM-C.(1)	BFM-C.(1) + ORTC	BFM-C.(1)	BFM-C.(1) + ORTC
1	20.38%	5.36%	59.40%	74.32%	37.56%	39.08%
2	27.36%	12.12%	45.00%	64.66%	37.31%	44.83%
3	29.45%	34.94%	67.60%	79.03%	56.60%	74.06%
4	49.08%	14.60%	59.44%	69.34%	41.25%	57.29%
5	40.27%	18.78%	53.02%	66.96%	37.11%	54.37%
6	45.90%	36.62%	58.97%	75.20%	56.64%	69.06%
7	41.35%	21.18%	50.25%	66.66%	40.75%	55.91%
8	−16.16%	16.19%	27.77%	53.76%	−57.74%	41.98%
9	−125.06%	9.44%	27.84%	52.71%	−53.52%	37.55%
10	−91.00%	6.55%	−41.00%	63.88%	−107.27%	53.31%
11	−141.89%	15.20%	−4.62%	62.54%	−182.52%	44.54%
12	−83.34%	9.81%	37.01%	59.18%	−29.64%	51.25%
13	−26.01%	13.90%	45.58%	62.53%	−29.64%	52.49%

ratio worsen as the tables become less "*homogeneous*". We tried to formalize such an intuition by running the algorithms on set of tables independently obtained by perturbing the entries of a real world one with a fixed probability. Table 12 provides the reduction ratios on the sets of routing tables artificially generated, grouped by cardinality and perturbing probability. We can notice that the

Table 12. Compression ratio on sets of routing tables obtained by perturbing the entries of real world one with fixed probability

#tables	Perturbating Prob.	BF-Multi	BF-Multi + ORTC	BFM	BFM + ORTC	BFM-C.(1)	BFM-C.(1) + ORTC
7	10%	33.71%	72.24%	75.23%	78.65%	25.58%	88.41%
15	10%	−50.15%	47.90%	35.00%	44.94%	−16.77%	62.09%
15	30%	−324.16%	−46.91%	3.14%	11.33%	−12.82%	13.68%

algorithms increase considerably the number of table entries when the homogeneity degree of the tables decreases, i.e. when the perturbing probability is higher.

Overall, the BFM and BFM-Cluster(1) heuristics present better performances than BF-Multi, with BFM-Cluster(1) being the best one.

Finally, concerning the IP lookup times, we have shown that in the auxiliary data structures the compression generally induces a proportional improvement. In fact, we have observed an evident lowering of the lookups times, as shown in Tables 7, 9 and 10. Again, the multibit and the binary search on IP length algorithms present a performance on the compressed tables better than the one obtained by the LC-Trie algorithm.

Many question are left open. First of all, an interesting issue is the extension of the experimentation work to the new IP release with addresses of 128 bits (IPv6). Moreover, it would be nice to determine other effective algorithms and heuristics, also guaranteeing better approximation ratios.

Finally, our work was meant as a first attempt toward the investigation of the effectiveness of the existing and newly proposed methods for IP tables compression. Starting from this basis, new heuristics should be devised also taking more into account other intrinsic features of the IP protocol, like for instance the hierarchical structure of the network.

Acknowledgement. The authors would like to thank Prof. Venkatachary Srinivasan for his help during the retrieval of the real world IP routing tables used in the tests and Prof. Marcel Waldvogel for his suggestions on the implementation of some auxiliary data structures.

References

1. Bilò, V., Flammini, M.: On the ip routing tables minimization with addresses reassignments. In: Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS) (2004)
2. Buchsbaum, A.L., Fowler, G.S., Krishnamurthy, B., Vo, K.-P., Wang, J.: Fast prefix matching of bounded strings. ACM Journal of Experimental Algorithms, vol. 8 (2003)
3. Crescenzi, P., Dardini, L., Grossi, R.: IP address lookup made fast and simple. In: Nešetřil, J. (ed.) ESA 1999. LNCS, vol. 1643, Springer, Heidelberg (1999)
4. Degermark, M., Brodnik, A., Carlsson, S., Pink, S.: Small forwarding tables for fast routing lookups. In: Proceedings of ACM Sigcomm, pp. 3–14 (1997)

5. Draves, R., King, C., Srinivasan, V., Zill, B.: Constructing optimal IP routing tables. In: Proceedings of The Conference on Computer Communications, 18th joint conference of the IEEE Computer and Communications Societies (INFOCOM) (1999)
6. Egevang, K., Francis, P.: The ip network address translator (NAT). Internet RFC 1631 (May 1994)
7. Ford, W., Topp, W.: Data Structures with C++. Prentice-Hall, Englewood Cliffs (1996)
8. Gupta, P., Lin, S., McKeown, N.: Routing lookups in hardware at memory access speeds. In: Proceedings of The Conference on Computer Communications, 17th joint conference of the IEEE Computer and Communications Societies (INFOCOM), pp. 1240–1247 (1998)
9. Gupta, P., Prabhakar, B., Boyd, S.P.: Near optimal routing lookups with bounded worst case performance. In: Proceedings of The Conference on Computer Communications, 19th joint conference of the IEEE Computer and Communications Societies (INFOCOM), pp. 1184–1192 (2000)
10. Kernen, T.: Traceroute org. http://www.traceroute.org/#Route Servers (2005)
11. Lampson, B., Srinivasan, V., Varghese, G.: IP lookups using multiway and multicolumn search. IEEE/ACM Transactions on Networking 7(3), 324–334 (1999)
12. NANOG. The north american network operators' group. http://www.nanog.org/lookingglass.html (2005)
13. Nilsson, S.: Home page. http://www.nada.kth.se/s̃nilsson
14. Nilsson, S., Karlsson, G.: Ip-address lookup using lc-tries. IEEE Journal of Selected Areas in Communications 17(6), 1083–1092 (1999)
15. Srinivasan, V.: Fast and efficient Internet lookups. PhD thesis, Washington University (1999)
16. Srinivasan, V., Varghese, G.: Faster ip lookups using controlled prefix expansion. ACM Transactions on Computer Systems 17(1), 1–40 (1999)
17. Michigan University and Merit Network. Internet performance and analysis (ipma) project. http://www.merit.edu
18. Waldvogel, M.: Fast Longest Prefix Matching: Algorithms, Analysis, and Applications. PhD thesis, Swiss Federal Institute of Technology - Zurich (2000)
19. Waldvogel, M., Varghese, G., Turner, J., Plattner, B.: Scalable high-speed ip routing lookups. In: Proceedings of ACM Sigcomm, pp. 25–36, (October 1997)

Algorithms for Longer OLED Lifetime

Friedrich Eisenbrand[1], Andreas Karrenbauer[2], and Chihao Xu[3]

[1] Fachbereich Mathematik, Universität Paderborn
eisen@math.uni-paderborn.de
[2] Max-Planck-Institut für Informatik, Saarbrücken
karrenba@mpi-inf.mpg.de
[3] Lehrstuhl für Mikroelektronik, Universität Saarbrücken
chihao.xu@lme.uni-saarland.de

Abstract. We consider an optimization problem arising in the design of controllers for OLED displays. Our objective is to minimize the amplitude of the electrical current flowing through the diodes which has a direct impact on the lifetime of such a display. The optimization problem consist of finding a decomposition of an image into subframes with special structural properties that allow the display driver to lower the stress on the diodes. For monochrome images, we present an algorithm that finds an optimal solution of this problem in quadratic time. Since we have to find a good solution in realtime, we consider an online version of the problem in which we have to take a decision for one row based on a constant number of rows in the lookahead. In this framework this algorithm has a tight competitive ratio. A generalization of this algorithm computes near optimal solutions of real-world instances in realtime.

1 Introduction

Organic Light Emitting Diodes (OLEDs) have received growing interest recently as more and more commercial products are equipped with such displays. Though they have many advantages over current technology like LCD, only small size OLED displays have entered the marked yet. One reason for this is the limited lifetime of those

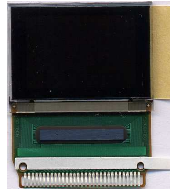


Fig. 1. Sample of a commercial OLED device with integrated driver chip

displays. While a lot of research is conducted on the material science side, the so-called *Multiline Addressing Scheme* for passive matrix OLED displays [7] tackles the lifetime-problem from an algorithmic point of view. It is based on the fact that equal rows can

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 338–351, 2007.
© Springer-Verlag Berlin Heidelberg 2007

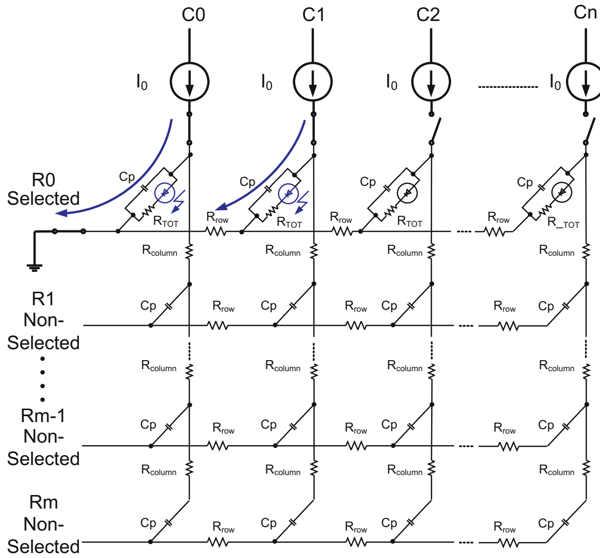


Fig. 2. Schematic electrical circuit of a display

be displayed simultaneously with a lower electrical current than in a serial manner. An explanation of this phenomenon can be found in [1] and [6]. Here we restrict ourselves to an informal description for self-containment.

A (passive matrix) OLED display has a matrix structure with n rows and m columns. At any crossover between a row and a column there is a vertical diode which works as a pixel. The image itself is given as an integral non-negative $n \times m$ matrix $(r_{ij}) \in \{0, \ldots, \varrho\}^{n \times m}$ representing its RGB values. Consider the contacts for the rows and columns as switches. For the time the switch of row i and column j is closed, an electrical current flows through the diode of pixel (i, j) and it shines. Hence, we can control the intensity of a pixel by the two quantities *electrical current* and *time*. In our application, the electrical current is equal for all pixels. Since high amplitudes of the electrical current or high peaks of intensity respectively, are the major issues with respect to the lifetime of the diodes [5], we try to trade as much time as possible for it. But since an image has to be displayed within a certain time frame T_f, it is a limited resource that we shall use as efficient as possible. Hence, the value r_{ij} determines the amount of time within the time frame in which the switches i and j have to be simultaneously closed. At a sufficient high frame rate e.g. 50 Hz, the perception by the eye is the average value of the light emitted by the pixel and one sees the image.

The traditional addressing scheme is row-by-row. This means that the switch for the first row is closed for a certain time while the switches for the columns are closed for the necessary amount of time dictated by the entries r_{1j}, $j = 1, \ldots, m$. Consequently the first row can be displayed in time $\max\{r_{1j} : j = 1, \ldots, m\}$. Then the second row is displayed and so on.

Consider the schematic image on the left of Fig. 3. Let us compute the amount of time which is necessary to display the image with this addressing scheme. The maximum value of the entries in the first row is 238. This is the amount of time which is necessary to display the first row. After that the second row is displayed in time 237. In total the time which is required to display the image is $238 + 237 + 234 + 232 + 229 = 1170$ time units.

Now consider the decomposition of the image as the sum of the three images on the right of Fig. 3. In the first image, each odd row is equal to its even successor. This means that we can close the switches for rows 1 and 2 simultaneously, and these two equal rows are displayed in 82 time units. Rows 3 and 4 can also be displayed simultaneously which shows that the first image on the right can be displayed in $82+41$ time units. The second image on the right can be displayed in $155+191$ time units while the third image has to be displayed traditionally. In total all three images, and thus the original image on the left via this decomposition, can be displayed in $82+41+155+191+156+38+38 = 701$ time units. This means that we could reduce the necessary time via this decomposition by roughly 40%. We could equally display the image in the original 1170 time units but reduce the peak intensity, or equally the maximum electrical current through a diode by roughly 40%.

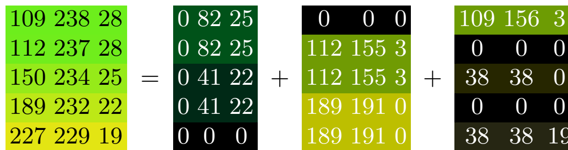


Fig. 3. An example decomposition

On real-world images, an optimal decomposition of the image allows a reduction of the electrical current to 56% on the average. This means an increase of lifetime by roughly 100%, see [5].

To benefit from this decomposition in practice, an algorithm to solve the optimization problem, which is formally described in Section 2, has to be implemented on a chip which is attached to the display, see Fig. 1. The following design criteria lie in the focus when engineering such an algorithm.

- The algorithm has to react in realtime.
- It must have low hardware complexity allowing small production costs.
- Consequently it has to rely only on a small amount of memory and it should be *fully combinatorial*, i.e. only additions, subtractions, and comparisons are used.

Especially the last of the above criteria clearly establishes a border between our approach and another technique [4] based on *Non-negative Matrix Factorization* [3,2].

Contributions of This Paper

First we show that monochrome images can be optimally decomposed in polynomial time. The presented algorithm has quadratic running time in the worst case. Therefore

we introduce an online version of this algorithm which takes a decision for one row, based on a lookahead of a certain fixed number of rows. This algorithm runs in linear time and has tight competitive ratio. On real world images it turns out that a lookahead of 3 rows gives the most satisfactory results when balancing approximation ratio, ease of implementation and running time. Our computational results show that this algorithm with a lookahead of 3 outperforms the previously best algorithm presented in [1] w.r.t. its practical approximation ratio and even more so w.r.t. its running time. This implies that nearly optimal Doubleline Addressing for real world images can be efficiently computed and, in particular, that an economic hardware implementation meeting the design criteria is possible.

2 The Formal Model

In this section, we will briefly review the formal model presented in [1]. Let $R = (r_{ij}) \in \{0, \ldots, \varrho\}^{n \times m}$ be the matrix representing the image. To decompose R we need to find matrices $F^{(1)} = (f_{ij}^{(1)})$ and $F^{(2)} = (f_{ij}^{(2)})$ where $F^{(1)}$ represents the singleline part and $F^{(2)}$ the two doubleline parts. More precisely, the i-th row of matrix $F^{(2)}$ represents the doubleline covering rows i and $i+1$. Since the overlay (addition) of the subframes must be equal to the original image to get a valid decomposition of R, the matrices $F^{(1)}$ and $F^{(2)}$ must fulfill the constraint $f_{ij}^{(1)} + f_{i-1,j}^{(2)} + f_{ij}^{(2)} = r_{ij}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, where we now and in the following use the convention to simply omit terms with indices running out of bounds. Since we cannot produce "negative" light we require also non-negativity of the variables $f_{ij}^{(\alpha)} \geq 0$. The goal is to find an integral decomposition that minimizes

$$\sum_{i=1}^{n} \max\{f_{ij}^{(1)} : 1 \leq j \leq m\} + \sum_{i=1}^{n-1} \max\{f_{ij}^{(2)} : 1 \leq j \leq m\} \ .$$

This problem can be formulated as an integer linear program by replacing the objective by $\sum_{i=1}^{n} u_i^{(1)} + \sum_{i=1}^{n-1} u_i^{(2)}$ and by adding the constraints $f_{ij}^{(\alpha)} \leq u_i^{(\alpha)}$. This yields

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} u_i^{(1)} + \sum_{i=1}^{n-1} u_i^{(2)} & & \\ \text{s.t.} \quad & f_{ij}^{(1)} + f_{i-1,j}^{(2)} + f_{ij}^{(2)} = r_{ij} & \text{for all } i, j & \quad (1) \\ & f_{ij}^{(\alpha)} \leq u_i^{(\alpha)} & \text{for all } i, j, \alpha & \quad (2) \\ & f_{ij}^{(\alpha)} \in \mathbb{Z}_{\geq 0} & \text{for all } i, j, \alpha & \quad (3) \end{aligned}$$

Note that the objective does not contain the f-variables.

Consider the constraints (1) for a fixed column j. By appending the constraint $0 = 0$ and by subtracting the $i-1$-st constraint from the i-th constraint, we obtain the following set of constraints

$$f_{ij}^{(1)} - f_{i-1,j}^{(1)} + f_{ij}^{(2)} - f_{i-2,j}^{(2)} = r_{ij} - r_{i-1,j} \qquad \text{for all } i, j. \qquad (4)$$

For each j the constraint-matrix is thus the node-arc incidence matrix corresponding to a graph like in Fig. 4. In the following we refer to this graph, which is solely determined by the number n of rows in the image, by the name *prototype displaygraph* $G_n = (V, A)$.

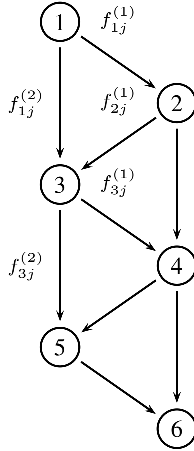


Fig. 4. Prototype displaygraph with variable-names of arcs entering and leaving row 3

The variables $f_{ij}^{(1)}$ correspond to the arcs going from left to right and vice versa. We call them arcs of type 1. The variables $f_{ij}^{(2)}$ are represented by the vertical arcs, called type 2. The number $r_{ij} - r_{i-1,j}$ is the *demand* $d_j(i)$ of vertex i in column j. The optimization problem can now be understood as follows.

> Given an integer matrix $R \in \mathbb{N}_0^{n \times m}$ reserve capacities $u : A \longrightarrow \mathbb{N}_0$ for the arcs A of G_n such that each of the demands d_j, $j = 1, \dots, m$ can be *individually* routed in G_n and such that $u(A) = \sum_{e \in A} u(e)$ is minimal.

In this context, *individually* routed means that for any column j the capacities admit a feasible flow satisfying the respective demands d_j.

3 Decomposing Monochrome Images in Polynomial Time

A *monochrome image* is an image $R \in \{0, 1\}^{n \times m}$. In this section we show that an optimal decomposition of such an image can be computed in polynomial time. The following example shows the transformation of an image into the demand matrix by the row operations that we described in the previous section.

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \rightsquigarrow \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \rightsquigarrow \begin{pmatrix} 1 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & -1 & -1 \end{pmatrix} \tag{5}$$

Each column is a $0, \pm 1$-vector. Furthermore it is easy to see that the occurrences of 1 and -1 in each column alternate and that each 1 is succeeded by a -1 and each -1 is preceded by a 1 disregarding the zeros inbetween. Moreover, the two matrices on the right of (5) have the same set of feasible solutions with respect to the capacities which are subsets of arcs such that the pairs of nodes $(1, 2)$, $(3, 4)$, $(2, 4)$, and $(1, 4)$ are connected in the corresponding subgraph. Therefore, we assume w.l.o.g. that each column yields exactly one such pair of nodes to which we also refer as a *commodity* in the following. In general the problem of optimally decomposing monochrome images can be understood as follows.

> Given commodities (s_j, t_j), $j = 1, \ldots, m$ with $s_j < t_j$ for each j and a number n, select a minimal number of arcs of G_n such that there exists a path from s_j to t_j for each $j = 1, \ldots, m$.

The nodes s_j are called *sources* and the nodes t_j are called *sinks*. Furthermore, if a node is neither a source nor a sink, we call it *Steiner*. The selection of arcs of G_n is given by a function $u : A \rightarrow \{0, 1\}$, where $u(a) = 1$ if the arc a is selected and $u(a) = 0$ otherwise.

The next lemma is easy to prove but crucial to obtain a polynomial-time algorithm. Here $u(\delta^{out}(i))$ denotes the number of selected arcs leaving node i. Similarly, $u(\delta^{in}(i))$ denotes the number of selected arcs entering i.

Lemma 1. *Given a feasible solution u, then there exists a feasible solution u' with*

$$u'(\delta^{out}(i)) \leq 1 \qquad \text{and} \qquad (6)$$
$$u'(\delta^{in}(i)) \leq 1$$

with the same total weight.

Proof. It is easy to see that we remain feasible if we substitute an arc of type 2 by the two arcs of type 1 incident to head and tail respectively. We do not change the number of selected arcs by selecting the other type 1 arc instead of the type 2 arc if $u(\delta^{out}(i)) > 1$ or $u(\delta^{in}(i)) > 1$ respectively as depicted in Fig. 5. Such a replacement is feasible, since each pair of nodes which was connected by a path before the replacement is still connected after the replacement.

In the forthcomming we maintain $u(\delta^{out}(i)) \leq 1$ and $u(\delta^{in}(i)) \leq 1$ as an invariant and call it *degree condition*. Thereby, the selection of one outgoing arc uniquely transforms the instance to the same problem with one row less. However, if we have selected the

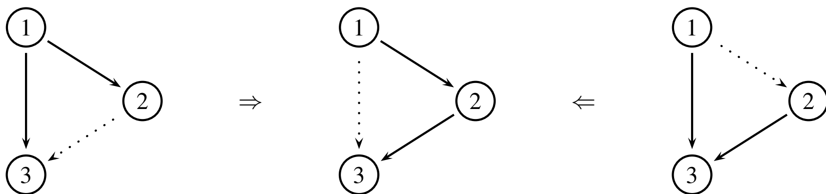


Fig. 5. Transformation to maintain the degree condition

outgoing arc of type 2 for node 1 and if node 2 is a source, we have to select the arc of type 2 as well to leave the second node to maintain the degree condition. In turn, this might force the same for the outgoing arcs of node 3 and so on. These implications evolve until either everything up to the current row is balanced or an odd commodity, say (s, t) with $t - s$ odd, produces a conflict (see Fig. 6). More precisely speaking, *balanced up to row* i means that the assignment to the capacities of the arcs a such that $head(a) \leq i$ is a feasible solution to the subinstance consisting of the rows $1, \ldots, i - 1$ of the given image. Hence, the solution of the subproblem starting at node i does not depend on how we have balanced up to row i. Note that both subproblems are considered with respect to the given image, i.e. all commodities (s_j, t_j) with $s_j < i < t_j$ are split into (s_j, i) and (i, t_j) where the former commodity is considered with the first subproblem and the latter with the second. It is easy to see that feasible solutions to the subproblems join to feasible solutions for the original problem. In particular, balanced up to row i implies that we may forbid the arc $(i - 1, i + 1)$ and remain feasible.

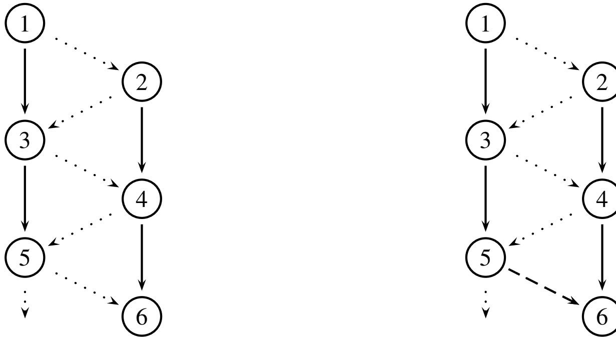


Fig. 6. In both examples, we are given the commodities $(1, 3)$ and $(2, 6)$. On the left, we additionally have $(3, 5)$ whereas on the right it is the odd commodity $(3, 6)$ instead. While the left example is balanced up to node 6, the commodity $(3, 6)$ produces a conflict on the right.

We will now present a basic dynamic programming scheme using node labels to store at node i how much it costs to balance up to node i. The label of the first node is 0 and all others are ∞ at the beginning. Let i be the current node. Assume that it is a source since otherwise we could simply skip it and proceed with the same label at node $i + 1$. We select arcs of type 2 until we either find a conflict or a node, say t, up to which we are balanced. If the label of i plus the number of selected arcs is smaller than the label of t, we update it accordingly. For a later reconstruction of the solution, we also store i as the predecessor of t. In case of a conflict we do not update anything. Afterwards, we proceed to node $i + 1$. If the label of $i + 1$ is more than 1 greater than label i, we also update it and set i as the predecessor of $i + 1$, i.e. selecting the arc of type 1. Then we repeat these steps until we reach the end. Because of the degree condition, we can transform any instance such that each node is the source of at most one commodity and also the sink of at most one commodity. Hence, by a preprocessing of the input data which takes $O(n \cdot m)$ time, we can annotate the nodes with the necessary information.

Since every step involves the visit of $O(n)$ nodes and arcs, the total time for computing the capacities is $O(n^2)$.

Theorem 1. *The optimal decomposition of an image given by a matrix $R \in \{0,1\}^{n\times m}$ can be computed in $O(n \cdot m + n^2)$.*

4 The Online Problem

Recall that we intend to develop an algorithm that finds a decomposition in realtime while keeping it simplistic enough such that it can be implemented on a chip with a low hardware complexity. Hence, we are looking for a linear time algorithm that uses only additions, subtractions, and comparisons. Since we do not want to scan over the whole rest of the graph in each iteration, it is natural to restrict the lookahead to a certain number of rows. It follows an online version of our problem where we have to fix the capacities of the outgoing arcs of a node only based on the knowledge of the following c rows. Again, we consider monochrome images first. At the end of this section we describe how our method can be adapted to decompose arbitrary colored images.

The canonical algorithm uses the one of Sec. 3 as follows. We solve the instance of the known c rows to optimality. According to that solution, we fix the outgoing arcs of the first node. After updating the instance and reading the next row, we repeat these steps until we reach the end. The computation takes $O(c \cdot n)$ time disregarding the time for the preprocessing that we have to spend anyways to parse the input.

In the following, we will first give a lower bound on the competitive ratio of any algorithm in that online setting. Afterwards, we will analyze the competitive ratio of the aforementioned approach. Before we state the theorem, it is helpful to have a look at following example where the adversary starts with the image in the middle and then reveals the fourth row according to the arc we have selected for the first node.

$$\begin{pmatrix} \not{1} & \not{0} & \not{0} \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \xleftarrow{\text{type 1}} \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ \star & \star & \star \end{pmatrix} \xrightarrow{\text{type 2}} \begin{pmatrix} \not{1} & \not{0} & \not{0} \\ \not{1} & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \tag{7}$$

The optimal value in both cases is 3. But after making the choice for the first row, the adversary force us to pay 4.

Theorem 2. *Any online algorithm that fixes the outgoing arcs of node i without knowing the rows $i + c$ and beyond, has a competitive ratio of at least $\frac{c+1}{c}$.*

Proof. An adversary reveals the first c nodes of an instance with the commodities (i, t_i) forall $i = 1, \ldots, c$ where $t_i \in \{c + 1, c + 2\}$ is chosen later depending which arc the algorithm picks following the idea shown in (7). If the algorithm selects the arc of type 1, then the adversary sets t_1 to the odd value. Otherwise, it is set to the even node. All other t_i are set such that the commodities (i, t_i) are even. The optimal solution of the residual problem is c. Hence, the achieved objective value is $c+1$ whereas the opposite choice for the first arc would yield an optimal solution of value c.

Theorem 3. *There is an algorithm with competitive ratio $\frac{c+1}{c}$.*

Proof. We first compare the optimal algorithm running on the complete instance with the one running on the first c rows. Let t be the node such that the selection of the type 2 arcs starting with the first node gets balanced with respect to the whole instance. If no such node exists, then in every feasible solution the arc of type 1 has to be picked for leaving node 1. This also holds for the instance restricted to the first c nodes. Note that the choice of the first arc only depends on the rows strictly less than t. Hence, if $t \leq c+1$, then the online algorithm makes the same decision on the first arc as the optimal one. Otherwise, it takes the arc of type 1. So let us assume that the arc of type 2 would have been the optimal choice. Hence, the optimal label of node t is $t-2$. On the other hand, choosing only arcs of type 1 yields a label of $t-1$. Since $t > c+1$, the ratio $\frac{t-1}{t-2} \leq \frac{c+1}{c}$. Since we can partition the solution that is found by the optimal algorithm into independent balanced parts, we can repeat these arguments on them.

4.1 A Compact $4/3$-Approximation

In this subsection, we unroll the generic algorithm for the case $c = 3$ and give a compact set of rules for the selection of the capacities. These rules can be generalized to decomposed colored images yielding a competitive approximation algorithm in practice. They are described as follows and depicted in Fig. 7.

Compact. We consider the first three nodes. If the first node is not a source, we skip it without selecting any outgoing arc. Assume it is a source in the following. If the corresponding sink is node 2 (see Fig. 7a), we select the arc of type 1. If node 2 is a *Steiner node* (Fig. 7b), i.e. it is neither a source nor a sink, then we select the arc of type 2. If node 2 is a source and node 3 is either the corresponding sink or Steiner (Fig. 7c/d), we select the arc of type 1. Otherwise, we select the arc of type 2 (Fig. 7e).

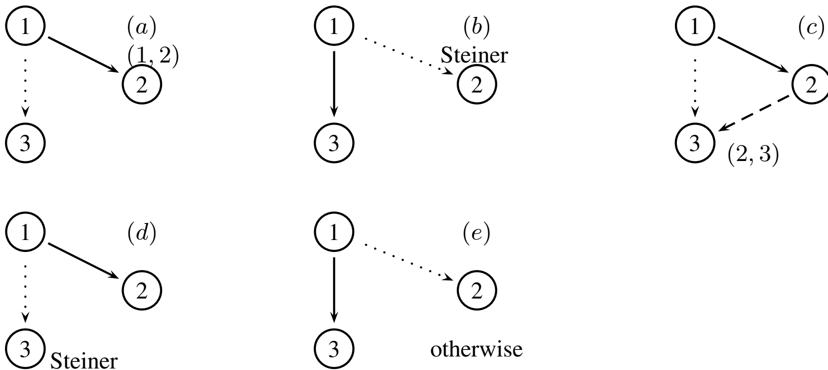


Fig. 7. The rules for the algorithm COMPACT

Lemma 2. *The algorithm* COMPACT *achieves a competitive ratio of* $4/3$.

Proof. The interesting cases are if node 1 and 2 are sources and their corresponding sinks are not revealed yet. Assume first that node 3 is Steiner. We show that we can transform every feasible solution such that the Steiner node is isolated (see Fig. 8). Consider a feasible solution where the arc between node 1 and 3 is picked. Since node 2 is a source the arc between 2 and 4 is also selected. Moreover there must be an arc between node 3 and 5. We can reconnect the tail of latter to node 4 and the head of the outgoing arc of node 1 to node 2. Thereby, we do not change the number of arcs and the routing remains feasible. If node 3 is a source instead, the demand of node 1 may go piggyback with the demand of node 2 or with the one of node 3. Since the first three nodes are sources, each of them has an outgoing arc in every feasible solution. If an adversary reveals that our decision to take the arc of type 2 for node 1 was wrong, we need one surplus arc to fix it. Until the adversary does not force us to change the parity, i.e. choose an arc of type 1, we do not use more arcs than optimal. Moreover, if we are forced to take such an arc, the problem decomposes into independent subproblems. Thereby, we use at most one surplus arc by three necessary ones and hence get a ratio of $4/3$.

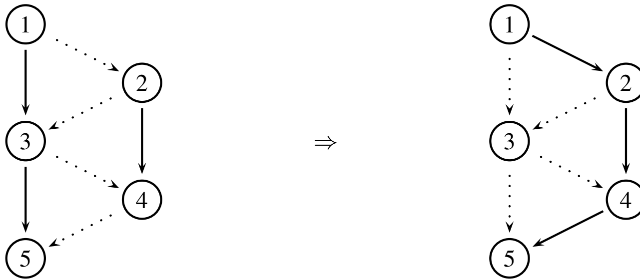


Fig. 8. Isolating a Steiner node

Generalizing to Colored Images. Recall that in the general case the instance is not given by a binary matrix but as $R = (r_{ij}) \in \{0, \ldots, \varrho\}^{n \times m}$. So we need to generalize our concepts for this purpose. We briefly sketch how this is done in our algorithm. Whenever $\max\{r_{ij} - r_{i-1,j} : 1 \leq j \leq m\} > 0$ we call the node i a source in the prototype displaygraph. For the ease of notation, we use the following abbreviation $\overline{r_i - r_{i-1}}$ for the maximum over all columns. Similarly, we call node i a sink whenever $\overline{r_{i-1} - r_i} > 0$. The degree condition transforms into $u(\delta^{out}(i)) \leq \overline{r_i}$ and $u(\delta^{in}(i)) \leq \overline{r_{i-1}}$. Similarly to the set of rules presented above, we define five rules for the general case. The rule a) for example translates into the rule in which we have to reserve a capacity of at least $\overline{r_2 - r_1}$ on the arc of type 1 leaving node 1. The other rules can be generalized accordingly. We do not know the exact approximation ratio of this generalized algorithm. In particular we do not know whether it exceeds $4/3$. However, as the computational results of the next section show, it behaves very well in practice.

5 Computational Results

We use the same testset and machine as in [1]. It is a Pentium M with 2GHz and 2MB L2 cache. The images are portraits of 197 employees of the MPI. They have a resolution of 180×240 pixels and a colordepth of 24 bit, i.e. $n = 180$, $m = 720$, and $\varrho = 255$. From [1], we take the algorithms called *ec-bf-mcgu-2* and *ec-bf-mcgu-4* which performed best there. They differ only by the fact that the former combines two rows and the latter combines up to four rows. We compare them to the generalized COMPACT algorithm (that solves the doubleline problem) and a cascading of it such that four or two rows may be combined. We will elaborate on the differences of *ec-bf-mcgu-4* and CASCADING to the doubleline addressing scheme at the end of this section.

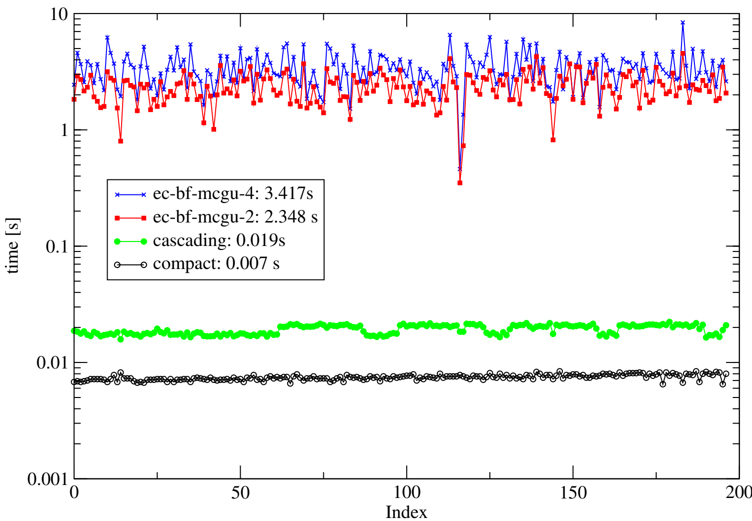


Fig. 9. Running time comparison of the old and new algorithms

In Fig. 9, we show the running times for each instance. The squares and the crosses (top) represent the old measurements of *ec-bf-mcgu-2* and *ec-bf-mcgu-4* respectively. Whereas the dots and circles (bottom) correspond to the new algorithms COMPACT and CASCADING. We connected the measurements by lines to guide the eye. Note the logarithmic scale of the time axis. One can see that the new algorithms are two orders of magnitudes faster than the old ones. Comparing the mean running times, the improvement is more than a factor of 300 between *ec-bf-mcgu-2* and COMPACT, and about 180 between *ec-bf-mcgu-4* and CASCADING. Moreover, the variance decreases drastically. This is due to the fact that the running time of the old algorithms depends strongly on the input data, i.e. on the unary encoding length, while it scales only with the size of the binary encoding length in the new ones.

It remains to show that the drastic improvements with respect to the running times are not at the cost of the approximation quality. Therefore, we solved the corresponding

integer linear programs for doubleline addressing with the commercial solver CPLEX. Thereby, we obtained the optimal solutions and were able to compare the per-instance approximation ratios of *ec-bf-mcgu-2* and COMPACT. The results are depicted in Fig 10.

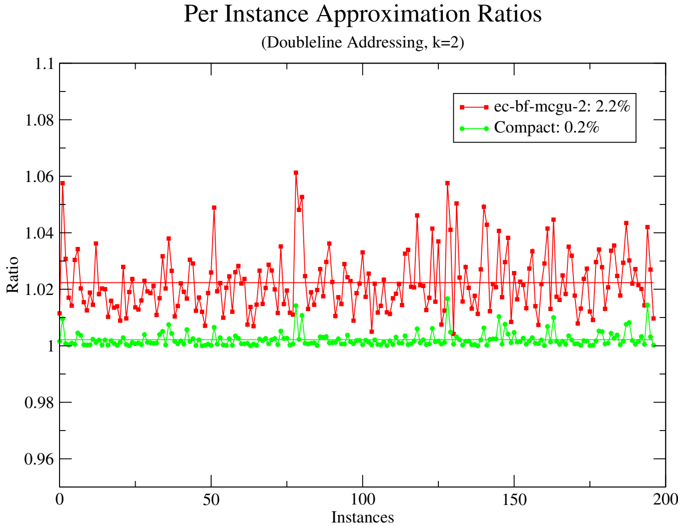


Fig. 10. Results of the old and the new doubleline algorithms normalized to the corresponding optimal solution. The horizontal lines indicate the respective means.

As one can see, the quality of the approximation of the new algorithm is not worse than for the one presented in [1]. In fact, on average it is even considerably better. Recall that the objective of our optimization problem is proportional to the electrical current and therefore has a direct impact on the lifetime of such a passive matrix OLED display. The average gap of 0.2% shows that we have found an algorithm that does not leave much room for improvement with respect to the necessary electrical current to display real world images using the doubleline addressing scheme.

However, one can consider combining more than two rows to reduce the electrical current even further. The heuristics of [1] have been implemented in such a more general way, that the number of lines up to which we want to combine them, is controlled by a parameter $k = 2, 3, 4, \ldots$ whereas COMPACT is specialized to the doubleline addressing scheme, i.e. $k = 2$. Nevertheless, doubleline addressing is an important building block for more advanced strategies. We outline here a simple one that is achieved by cascading COMPACT. This means that we take the two frames that contain the computed doublelines and feed both independently as input to COMPACT again. Thereby two doublelines of the outcome of the first phase may potentially be combined to a doubleline which represents the combination of four lines with respect to the original image. Thereby, we push forward into the range of *ec-bf-mcgu-4* without considering the combination of three lines. The ratios of the objectives of CASCADING and *ec-bf-mcgu-4* for each instance are presented in Fig. 11.

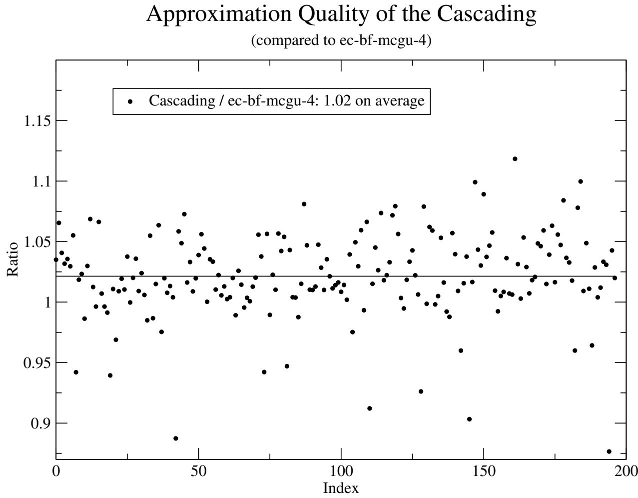


Fig. 11. The relative objective of CASCADING with respect to *ec-bf-mcgu-4*. The horizontal line indicates the average ratio of 1.02.

The slightly worse average approximation ratio by a factor of 1.02 is more than compensated by the improvement with respect to the running time by a factor of about 180. Moreover, it is not possible that CASCADING yields a worse objective value than COMPACT on the same instance whereas this behavior occurred on some instances concerning *ec-bf-mcgu-2* and *ec-bf-mcgu-4*. However, there might be other strategies that are simplistic enough to guarantee a fast running time at low hardware complexity to close the gap. This is subject to ongoing research.

We want to conclude with a brief discussion of the applicability of consecutive Multiline Addressing to a broader set of images. Based on the results for human faces, it is natural to ask for photographs in general. It turned out that on a variety of over 3000 pictures, COMPACT achieves a reduction of the electrical current to 56% on the average with a mean per instance approximation ratio of 1.003 compared to the optimal solution provided by CPLEX. The results for two exemplary music videos are even better with a reduction to 51% which is only a factor of 1.002 away from the optimum. We explain this behavior by the fact that the content of photos is rather smooth, e.g. they are not dominated by sharp edges as in artificial images like cliparts and text by bitmap fonts. This is in agreement with the results on a testset of wallpapers for mobile phones with a mean reduction to 63% and approximation factor of 1.005 averaged over about 4500 samples. We observed that diagonal lines, in particular if the width is only one pixel and the contrast to the neighborhood is high, constitute an obstacle to Multiline Addressing.

Finally, we want to thank Markus Tetzlaff for providing us with the large set of his digital photos and Tobias Jung for performing the tests as part of his bachelor thesis.

References

1. Eisenbrand, F., Karrenbauer, A., Skutella, M., Xu, C.: Multiline Addressing by Network Flow. In: Erlebach, T., Azar, Y. (eds.) ESA 2006. LNCS, vol. 4168, pp. 744–755. Springer, Heidelberg (2006)
2. Lee, D.D., Seung, H.S.: Algorithms for non-negative matrix factorization. Advances in Neural Information Processing Systems 13 (2001)
3. Paatero, P., Tapper, U.: Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values. Environmetrics 5, 111–126 (1994)
4. Smith, E., Routley, P., Foden, C.: Processing digital data using non-negative matrix factorization. Patent GB 2421604A, pending (2005)
5. Soh, K.M., Xu, C., Hitzelberger, C.: Dependence of OLED Display Degradation on Driving Conditions. In: Proceedings of SID Mid Europe Chapter Fall Meeting (2006)
6. Xu, C., Karrenbauer, A., Soh, K.M., Wahl, J.: A New Addressing Scheme for PM OLED Display. Society for Information Display (SID) Symposium Digest (2007)
7. Xu, C., Wahl, J., Eisenbrand, F., Karrenbauer, A., Soh, K.M., Hitzelberger, C.: Verfahren zur Ansteuerung von Matrixanzeigen. Patent 10 2005 063 159, Germany, pending (2005)

Improving Tree Search in Phylogenetic Reconstruction from Genome Rearrangement Data

Fei Ye[1], Yan Guo[2], Andrew Lawson[1], and Jijun Tang[2,⋆]

[1] Department of Epidemiology and Biostatistics
University of South Carolina
Columbia, SC 29208, USA
[2] Department of Computer Science & Engineering
University of South Carolina
Columbia, SC 29208, USA
jtang@cse.sc.edu

Abstract. A major task in evolutionary biology is to determine the ancestral relationships among the known species, a process generally referred as phylogenetic reconstruction. In the past decade, a new type of data based on genome rearrangements has attracted increasing attention from both biologists and computer scientists. Methods for reconstructing phylogeny based on genome rearrangement data include distance-based methods, direct optimization methods (GRAPPA and MGR), and Markov Chain Monte Carlo (MCMC) methods (Badger). Extensive testing on simulated and biological datasets showed that the latter three methods are currently the best methods for genome rearrangement phylogeny. However, all these tools are dealing with extremely large searching spaces; the total number of possible trees grows exponentially when the number of genomes increases and makes it computationally very expensive. Various heuristics are used to explore the tree space but with no guarantee of optimum being found. In this paper, we present a new method to efficiently search the large tree space. This method is motivated by the concept of particle filtration (also known as Sequential Monte Carlo), which was originally proposed to boost the efficiency of MCMC methods on massive data. We tested and compared this new method on simulated datasets in different scenarios. The results show that the new method achieves a significant improvement in efficiency, while still retains very high topological accuracy.

1 Introduction

The goal of phylogenetic analysis is to determine the evolutionary relationships among organisms and their genomes. A *phylogeny* for a set of N genomes (species) is a preferably N leaves binary tree, with each leaf labeled by a distinct element of the input set. Fig 1 shows two proposed phylogenies [18], the left one is for 12 species of the Campanulaceae (bluebell flowers) family (with Tobacco as an outgroup), represented in the form of *cladogram*; and the right one is for herpesviruses that are known to affect humans, represented in the form of an *unrooted* tree.

⋆ Corresponding author.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 352–364, 2007.
© Springer-Verlag Berlin Heidelberg 2007

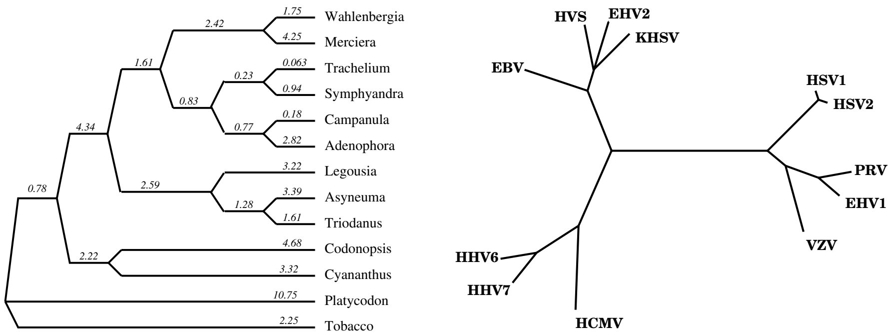


Fig. 1. Phylogenies for Campanulaceae (left) and herpesviruses (right)

To date, DNA (or protein) sequence data is the primary source of information for phylogenetic analysis. In the last decade, genome rearrangement (also known as gene-order) data are emerging into the field and many researchers showed great interests about it [8,20,21,24].

Biologists can infer the ordering and strandedness of genes on a chromosome and thus represent each chromosome by an ordering of signed genes (where the sign indicates the strand). These gene orders can be rearranged by evolutionary events such as inversions and transpositions. The relative rarity of genomic rearrangements, together with the increasing availability of complete genome sequences, make them very attractive as new sources for genome comparison. Developing appropriate tools for analyzing such data is therefore an important area of research. During the past several years, computer scientists have been able to make substantial progress in genome rearrangement research. With the solution for inversion distance [12] and inversion median [4], we were able to estimate phylogenies and ancestral genomes based on inversions (the dominant events in organellar genomes).

There are several widely used methods for genome rearrangement analysis, including neighbor-joining [25], `GRAPPA` [16], `MGR` [2] and `Badger` [14]. Using the later three generally will achieve better accuracy than using distanced based methods such as neighbor-joining. However, since all these three methods need to find the best tree from a large number of possible trees (tree space), they all face the similar problem of scalability: none of these three methods can be used for more than 15 genomes (species) because the total number of possible trees increases exponentially with the number of genomes[1]: there are 13 billion trees for 10 genomes, more than $7,000$ billion trees for 15 genomes, and more than 2^{67} trees for merely 20 genomes. Using today's most powerful machines, `GRAPPA` (the fastest among the three) can finish searching the tree space for 15 genomes within several minutes, but such search will take more than $6,000$ centuries for 20 genomes. Even if the CPU power continues to increase under Moore's law (doubled for every 18 months), such computation will still take at least 20 years.

[1] The number of all possible binary trees is $(2N-5)!! = (2N-5) \times (2N-7) \cdots \times 3$ for N genomes [10].

One way to remedy this problem is to use disk-covering methods (DCM), introduced by Warnow and her group. Tang et al. combined the DCM1 [13] approach with GRAPPA, and produced DCM-GRAPPA [28], which can analyze datasets of up to 1,000 taxa with high accuracy. DCM-GRAPPA works in three steps: it first decomposes the dataset into smaller overlapping subproblems, then runs GRAPPA as the base method on these subproblems to obtain subtrees, and finally combines the subtrees to build a tree for the original dataset. Since GRAPPA has a limit on the number of genomes it can handle, DCM-GRAPPA has to recursively call DCM until each subproblem size falls below that limit (currently set to 13 genomes). Because the threshold is small, large problems require many levels of recursive decomposition, which is not only time-consuming but also risks propagating and amplifying error in the assembly of the subtrees. For $1,000$ genomes, it generally requires $6-7$ level of recursively calls if the maximum subproblem size is limited to 13, but only $2-3$ levels when the limit is raised to 20. As a result, improving the efficiency of GRAPPA is still desirable.

Another standard approach to scaling is to compute the smallest possible nontrivial trees: quartet trees, defined on just four taxa. Quartet methods rely on finding the optimal 4-leaf tree for each quartet and using this information to build the overall tree. Liu et al. developed an optimization algorithm [15] for the NP-hard problem of computing optimal trees for each quartet, as well as a variation of the dyadic method [9] to choose suitable short quartets. This method is able to handle $20-30$ genomes. However, it begins to loose accuracy when the input dataset has more than 18 genomes, hence is not suitable to be used as a base method for DCM-GRAPPA.

In the past five years, facing similar problem in Bayesian analysis of massive datasets, statisticians developed particle filtration techniques (also known as sequential Monte Carlo methods) [7,23] to improve the efficiency in MCMC samplings. In this paper, we present a new tree search method motivated by particle filtrations. After some background review and definitions, we describe in Section 5 our new tree search method in detail; in Section 6, we evaluate the new method on simulated datasets. The results suggest that our method is much faster than GRAPPA, with very limited loss of accuracy. This new method can be integrated with DCM methods to further improve the analysis of large scale datasets.

2 Backgrounds

2.1 Genome Rearrangements

We assume a reference set of n genes $\{g_1, g_2, \cdots, g_n\}$, thus a genome can be represented as a signed ordering of these genes, and each gene is given an orientation that is either positive, written g_i, or negative, written $-g_i$. Genomes can evolve through events such as inversions, transpositions and transversion, as well as many other events.

Let G be the genome with signed ordering of $g_1, g_2, \cdots, g_n$. An *inversion* between indices i and j $(i \leq j)$, transforms G to a new genome with linear ordering

$$g_1, g_2, \cdots, g_{i-1}, -g_j, -g_{j-1}, \cdots, -g_i, g_{j+1}, \cdots, g_n$$

A *transposition* on genome G acts on three indices i, j, k, with $i \leq j$ and $k \notin [i, j]$, picking up the interval $g_i, g_{i+1}, \cdots, g_j$ and inserting it immediately after g_k. Thus genome G is replaced by (assume $k > j$):

$$g_1, \cdots, g_{i-1}, g_{j+1}, \cdots, g_k, g_i, g_{i+1}, \cdots, g_j, g_{k+1}, \cdots, g_n$$

An *transversion* is a transposition followed by an inversion of the transposed subsequence; it is also called an *inverted transposition*.

The *edit distance* between two genomes is the minimum number of evolutionary events required to transform one genome into the other. When only inversions are allowed, the edit distance is the *inversion distance*. The *score* of a tree is the sum of the costs of its edges, where the cost of an edge can be defined as the distance between the two genomes that label the endpoints of the edge. Finding the (minimum) score of a tree generally requires the determination of gene orders on each internal node, which is itself very difficult. For a three-leave tree, the tree score can be obtained by finding a genome that minimizes the sum of pairwise distances between itself and each of the three leaf genomes. Such procedure is generally referred as *median problem of three*, or *median problem* for short, which is NP-hard [3,22] when inversion distances are used.

2.2 Phylogenetic Reconstruction from Genome Rearrangements

Methods for phylogeny analysis based on genome rearrangement data include distance-based methods (for example, neighbor-joining [25]), maximum parsimony methods based on encodings [31], and direct optimization methods. The latter, pioneered by Sankoff et al. [26] in their package `BPAnalysis` and improved by `GRAPPA` [16] and `MGR` [2], are the most accurate. A Markov Chain Monte Carlo method (`Badger`) [14] developed by Larget et al. is also widely used with comparable accuracy. The reconstruction goal of both `GRAPPA` and `MGR` is to find the tree(s) with the lowest score. One should note that such minimum score tree may not be the true phylogeny–it is just an estimation of the history. In deed, all phylogenetic methods so far cannot guarantee that the true phylogeny be found, thus the accuracy of a method should be carefully assessed using both simulated and biological data. Besides returning a phylogeny, all these methods can also give an estimate of ancestral genomes, which will have great utility for biologists interested in the process of genome rearrangement.

2.3 GRAPPA

`GRAPPA` is an exhaustive search method, moving systematically through the space of all $(2N-5)(2N-7)\cdots 3$ possible trees on N genomes. For each tree, the program tests a lower bound to determine whether the tree is worth scoring; if so, then the program will determine the tree score by iteratively solving the median problems at internal nodes until convergence, as outlined in Fig. 2.

The speed of `GRAPPA` is regulated by two factors: the efficiency of median computation and the pruning rates, i.e. how many trees can be discarded before being scored. Moret et al. developed several lower bounds based on triangular inequalities [17]. All these bounds are very tight and easy to compute (much easier than the scoring procedure). As a result, more than 99.99% trees can be discarded without being scored. Further speed-up is achieved by using a branch-and-bound approach that can discard most trees without even generating them. Overall, `GRAPPA` achieves a billion-fold speed up over its predecessor of `BPAnalysis` and can finish the original 13-genome *Campanulaceae* dataset [6] within 20 minutes on a single workstation. Even with such speedup,

```
Initially label all internal nodes with gene orders
Repeat
    For each internal node v, with neighbors A, B and C, do
    Solve median problem on A, B, C to yield m
    If relabeling v with m improves the tree score, then do it
Until no change occurs
```

Fig. 2. The GRAPPA scoring procedure

GRAPPA is not suitable for datasets with more than 15 genomes. Both MGR and Badger face similar limitations [2,14].

3 Searching the Large Tree Space

GRAPPA is not the only package that faces the dilemma of dealing with the fast growing tree space. Indeed, except for distance based methods (such as neighbor-joining [25]), all popular phylogeny packages–including those developed for DNA sequence data–have to search this space for phylogenies. For dataset with many species, people have to rely on various heuristics to explore the tree space with no guarantee that the optimal trees will be found.

Three natural approaches for searching the tree space are nearest-neighbor interchanges (NNI), subtree pruning and regrafting (SPR), and Tree-Bisection- Reconnection (TBR) [27]. In NNI, one of the internal edges is chosen at random and the four subtrees (by removing the edge and its two nodes) are reconnected randomly. In SPR, a random edge is selected and two subtrees are created, then one of the two subtrees is removed at random and reinserted along a random edge in the other subtree. In TBR, similar to SPR, one edge is removed and the tree is divided into two subtrees, then they are joined by an edge connecting two midpoints of edges of the two subtrees. Figure 3 shows two examples of these heuristics.

These methods form the core of many phylogeny analysis packages, including Bayesian methods such as Badger. Since there is no guarantee of convergence by using these methods, various schemes (such as choosing multiple starting points) are developed with the hope that the search will converge. However, Mossel et al. [19] recently proved that many of the popular Markov Chain Monte Carlo methods using the above tree searching techniques take exponentially long time to converge.

On the other hand, GRAPPA uses an exhaustive approach to examine all possible trees. In order to perform this exhaustive search, a fast tree generating procedure using depth-first approach is implemented, which in turn provides a new way of defining the tree space.

Let's examine the depth-first procedure in detail. Assuming the tree generating procedure works on levels, and each level k has all the partial trees containing the first k genomes. We pick the first three genomes (genome 1, 2 and 3) and create the (only) binary tree in level 3. named $T_{3\cdot 1}$ (tree number one in level 3). Then, we will attach the next genome (genome 4) to the first edge of this tree, and generate tree $T_{4\cdot 1}$. We repeatedly add the next genomes until it reaches the last level (genome N is included), resulted in the first complete N-genome tree ($T_{N\cdot 1}$) being generated. The next tree ($T_{N\cdot 2}$)

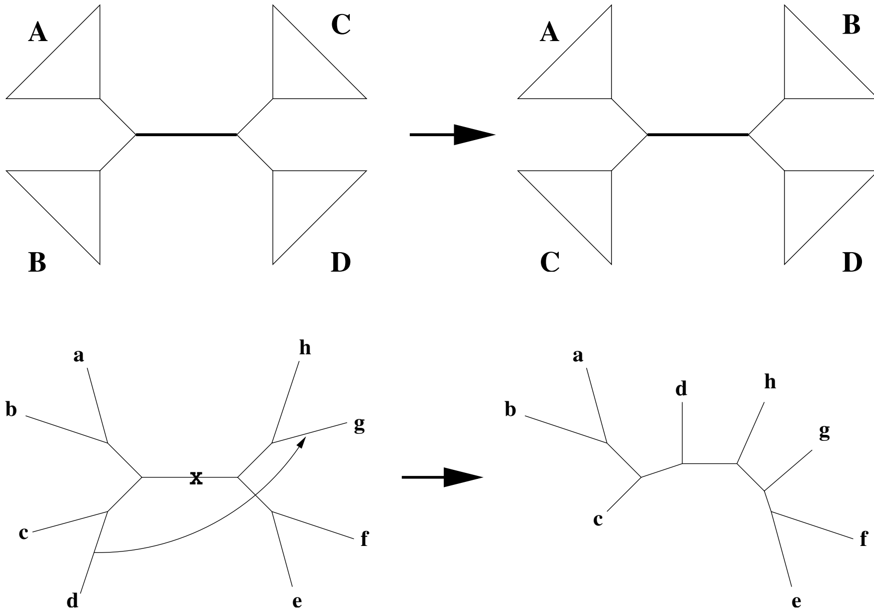


Fig. 3. Examples of NNI (top) and TBR (bottom)

is generated by attaching genome N to the second edge of $T_{N-1\cdot 1}$, and tree ($T_{N\cdot 2N-5}$) is generated by attaching genome N to the last edge of $T_{N-1\cdot 1}$. The depth-first search has reached the depth, so it will move a level up and attach genome $N-1$ to the second edge of tree $T_{N-2\cdot 2}$, and generates the next block of $2N-5$ trees for N genomes. We then move up and down the levels as in the standard depth-first search, until all trees are generated.

We can define a tree space by assigning every tree a unique tree number with respect to the ordering of its appearance in the depth-first generating procedure. This space has a unique property that trees with close tree numbers are *generally* similar in topology. For example, the first $2N-5$ trees has difference of only one edge. The branch-and-bound method developed in GRAPPA [29] was based on this tree generating procedure and is the fastest options for using GRAPPA.

GRAPPA also provides a stepping function to quickly explore the tree space: if user sets the stepping interval of S, then it will only generate trees with number $1+S, 1+2S, \ldots$ as defined in the above space. Of course, the problem is obvious: it has a high probability of missing the best tree if the stepping is set too high, and is too slow if such value is small. The exhaustive search approach can be viewed as a special case when the stepping interval is set to one.

4 Particle Filtration Technique

In the context of sequential estimation, full MCMC algorithms must re-sample when new data arrives, hence the parameter and data spaces are enlarged to fit the new

situation, which is computationally very expensive when the data space is large. Sequential Monte Carlo methods (or particle filtration) techniques were developed to deal with such data-parameter enlargement problems.

Particle filtration techniques can be applied to the situation where subsets of the target data space are progressively included in the analysis [5,23]. They are also used to deal with massive non-sequential datasets where the data space can be divided into subsets and sequentially included into the analysis. The method starts by initializing a set of parameter estimates and uses importance sampling to filter out the particles, i.e., the parameters that have the least posterior probability after incorporating the additional data, by reweighing the parameter estimates derived from the current posterior distribution, using importance sampling. With this approach, the time saving can be dramatic. In addition, particle filtration only requires the adoption of a single step Metropolis update within re-sampling steps that are guaranteed to come from the posterior distribution and hence the usual convergence requirement does not apply to it.

Although particle filtration is developed for MCMC sampling on massive data and therefore requires a definition of posterior distribution, the idea of particle filtration, especially the concept of importance sampling and re-sampling scheme, can be adopted by any phylogeny packages to explore the large tree space.

5 The New Algorithm

5.1 Overview

Our new algorithm integrates particle filtration with the various tree search techniques discussed in Section 3. Let D denote the whole tree topology space for N genomes, and let D_1 denote a subset of trees from D, which is served as the initial trees for updating.

The algorithm of our method can be described with the following steps:

1. (Loading) Load a subset of D_1 trees into the memory. These trees can be randomly picked from the whole phylogenetic tree space.
2. (Resampling) For every tree D_{1i} in D_1, search its neighborhood in the tree space. If a tree with higher weight exists, replace the current D_{1i} tree with this one.
3. (Updating) After getting a new set of D_1, we then call the MCMC updating methods (such as NNI, TBR and SPR) to update the D_1 trees. The update will stop after certain number of updates are performed.
4. (Reporting) Choose the best tree found in the previous steps, and return the result.

5.2 Implementation Details

The above procedures are standard in particle filtration methods. However, there are a few modifications in each step since the new algorithm does not handle posterior distributions.

In step one, all particle filtration methods require loading as much as possible data into the initial subset (D_1). Since the tree space is too large, the portion of trees can

be loaded into memory is always a small percentage. In our experiments, we found that the final results are not very sensitive to the size of D_1 (denote $|D_1|$), and only a few hundred trees are needed in forming the initial D_1. In fact, selecting which trees to form set D_1 is more important. Our strategy is to divide the whole tree space into $|D_1|$ subspaces (using tree numbers) and therefore each subspace contains $\frac{(2N-5)!!}{|D_1|}$ trees. Within each subspace, we then randomly select one tree into D_1. The purpose of doing so is to go deep enough into the tree space so that the trees are not chosen only from a small portion of the whole dataset. Since the tree number can be very large, we use the GNU Multiple Precision Arithmetic Library (GMP) to handle large integer numbers required in this step.

In the re-sampling procedure (step 2), trees in D_1 are updated by comparing the importance weight of each tree, i.e., a tree with higher weight will substitute the old tree in D_1. Such weight is originally defined to be proportional to the posterior density. We define a weight that is proportional to it inversion tree score, and a tree with lower tree score is given higher weight. Such score can be computed by the GRAPPA scoring procedure, or it can be estimated using the linear programming method [30] if the median is too difficult to compute. The searching for new trees should be done locally, which is utilized by using the stepping function in GRAPPA. Specifically, the local search is controlled by stepping interval S and number of steps U_1, all can be kept in the range of several hundred.

In standard particle filtration methods, the updating procedure (step 3) is performed according to the importance weight of each tree in D_1, i.e., trees with higher weight will have higher chances to be updated. In our current implementation, we choose to keep it simple and all trees in D_1 are updated exactly the same number of times, denoted as C (number of *cycles*). As we can see in the experimental results, we only need a small number of cycles to generate very satisfying results. MCMC methods are requested for this update, thus all the standard MCMC update methods like NNI, TBR and SPR can be used. However, since SPR and NNI are localized updates compared to TBR, and a local search has already been done in step 2, we only use TBR in the new method.

6 Experimental Results

6.1 Setup of Simulations

We set out to examine the performance (in terms of speed and accuracy) of the new method. We concentrated our experiments on simulated datasets because topological accuracy can be easily assessed when the true trees are known, using measurements such as *false negative* and *false positive*. Let T be a tree leaf-labelled by a set of genomes, deleting some edge e from T produces a bipartition which splits the genomes into two sets. Let T be the true tree and let T' be the inferred tree; then the false negatives (FN) are those bipartitions that appear in the true tree T but do not appear in the inferred tree T'. Similarly, false positives (FP) are those bipartitions that appear in T but not in T' [17]. The goal of all phylogeny methods is to obtain both lower false

negative and false positive. In our simulations, since all true trees and reconstructed trees are binary trees, thus $FP = FN$, and we only report FN here. The rate of FN is defined as the number of false edges divided by the number of internal edges of the true tree ($N-2$ for N genomes). A rate of smaller than 5% is generally considered acceptable [27].

We generated datasets of 14, 16, 18 and 20 genomes with 100 genes for each genome (approximately the size of small organelle genomes). We used various number of evolutionary rates: letting r denote the expected number of evolutionary events along an edge of the true tree, we used values of r in the range of 2 to 12. The actual number of events along each edge is sampled from a uniform distribution on the set $\{\frac{r}{2},\ldots,\frac{3r}{2}\}$. While all computations were based on inversion distances and inversion medians, we generated the data with a deliberate model mismatch to test the robustness of the methods, using a mix of 80% inversions and 20% transpositions. For each combination of parameter settings, we ran 10 datasets and averaged the results. All the experiments are conducted on a Linux cluster with 152 Intel Xeon CPUs, but each CPU works independently on a test task.

Since GRAPPA can not handle more than 15 genomes, we only tested GRAPPA on datasets with 14 genomes and compared its accuracy with the new method. For GRAPPA, each dataset was tested with the branch-and-bound method, which is the fastest version available. For the new method, there are several parameters that have impact on the results: $|D_1|$ (number of trees in $D1$), U_1 (number of steps in re-sampling), S (stepping intervals) and C (number of TBR updates). In our experiments, we found that C has the biggest impact on both speed and accuracy, thus we only tested different C values (100, 300 and 500) for each dataset while the other variables were fixed. Since the tree space of 20 genomes is very large, we also tested those datasets with $C = 1000$. The choice of other parameters is listed in table 1:

Table 1. Parameters in the experiments

number of genomes	$\lvert D_1\rvert$	U_1	S
14	200	200	400
16	400	200	4000
18	600	400	40000
20	1000	1000	400000

6.2 Topological Accuracy

Fig 4 shows the accuracy in term of average FN rate, each graph contains results from different number of TBR updates C. Not surprisingly, the accuracy increases with the number of TBR updates, and using $500-1000$ cycles of such updates produced results with very high accuracy (with less than 5% errors). Fig 4 (a) also shows the result obtained from exhaustive search using the original GRAPPA, which is almost identical to those obtained from the new method.

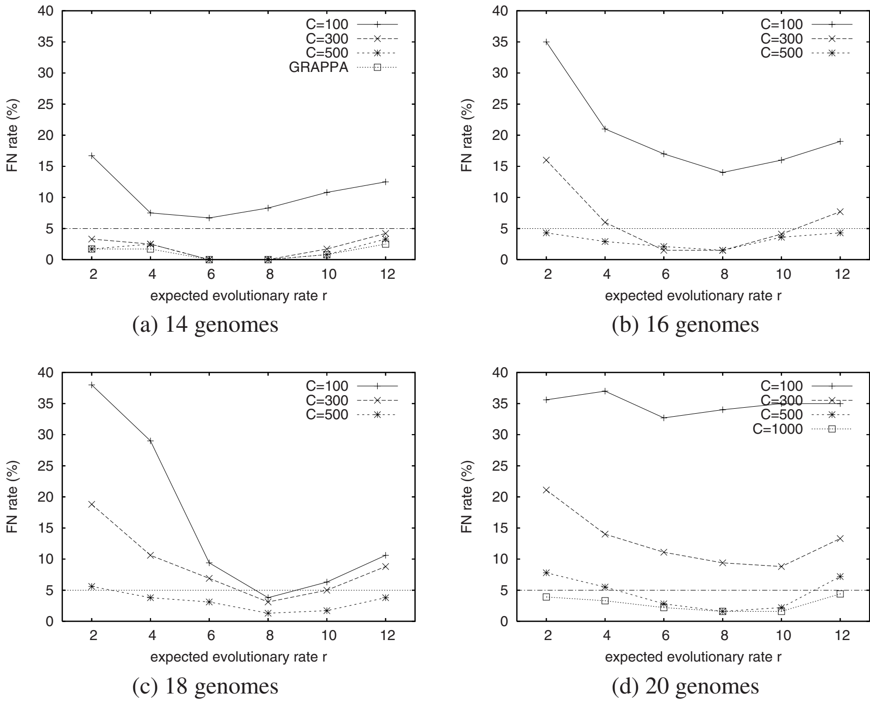


Fig. 4. Average FN rates as a function of r and number of cycles C

Fig 5 shows the accuracy results of using 500 – 1000 TBR updates. Compared to the results showed in [15], this figure suggests that this new method is more accurate than the quartet methods, thus can be used as a better base method for disk-covering methods.

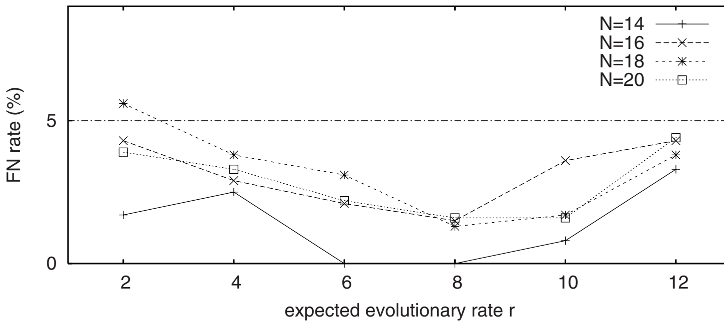
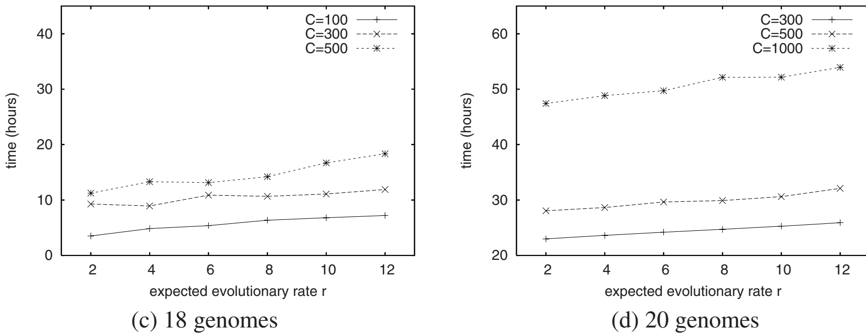


Fig. 5. Best average FN rates as a function of r and N

6.3 Speed

Our method is also very fast–it took only $1 \sim 2$ days to compute 20 genomes, instead of $6,000$ centuries as we projected by using GRAPPA. In other words, we achieved ten billion-fold speed-up while still retain very high accuracy. Fig 6 shows the average time spent for 18 and 20 genomes. Roughly speaking, the number of trees examined needs to be doubled when number of genomes N increases by one, thus the new method scales better than using exhaustive approach and has the potential to handle several more genomes.



(c) 18 genomes (d) 20 genomes

Fig. 6. Average running time as a function of r and number of cycles C

7 Conclusions

This paper presents a new method to search the large phylogeny space based on the concept of particle filtration. Although this method is now presented along with GRAPPA, such technique can be easily applied to other phylogeny packages. Since this method is originally designed to extend the range of base methods for DCMs, we will further assess the impact of using it over the direct use of GRAPPA as a base method.

This paper verifies the particle filtration to be a powerful tool in searching large space. There are many research subjects in Bioinformatics that face similar problem. For example, the median computation in genome rearrangement analysis needs to explore the space consists of all possible permutations–for signed genomes with n genes, there are $2^n n!$ possible permutations. The fact that inversion median problem is in APX reflects the difficulty of searching such large space. Applying particle filtration may provide a more efficient method for the median problems.

Acknowledgments

The authors were supported by US National Institutes of Health (NIH grant number R01 GM078991-01) and by the University of South Carolina.

References

1. Blanchette, M., Sankoff, D.: The median problem for breakpoints in comparative genomics. In: Jiang, T., Lee, D.T. (eds.) COCOON 1997. LNCS, vol. 1276, pp. 251–263. Springer, Heidelberg (1997)
2. Bourque, G., Pevzner, P.: Genome-scale evolution: Reconstructing gene orders in the ancestral species. Genome Research 12, 26–36 (2002)
3. Caprara, A.: Formulations and hardness of multiple sorting by reversals. In: RECOMB'99. Proc. 3rd Int'l Conf. on Comput. Mol. Biol, pp. 84–93. ACM Press, New York, NY, USA (1999)
4. Caprara, A.: On the practical solution of the reversal median problem. In: Gascuel, O., Moret, B.M.E. (eds.) WABI 2001. LNCS, vol. 2149, pp. 238–251. Springer, Heidelberg (2001)
5. Chopin, N.: A sequential particle filter method for static models. Biometrika 89, 539–552 (2002)
6. Cosner, M.E., Raubeson, L.A., Jansen, R.K.: Chloroplast DNA rearrangements in Campanulaceae: Phylogenetic utility of highly rearranged genomes. BMC Evol. Biol. vol. 4(27) (2004)
7. Doucet, A., de Freitas, N., Gordon, N.: Sequential Monte Carlo Methods in Practice. Springer, Heidelberg (2001)
8. Downie, S., Palmer, J.: Use of chloroplast DNA rearrangements in reconstructing plant phylogeny. In: Soltis, P., Soltis, D., Doyle, J. (eds.) Plant Molecular Systematics, pp. 14–35 (1992)
9. Erdős, P.L., Steel, M.A., Székely, L.A., Warnow, T.: Local quartet splits of a binary tree infer all quartet splits via one dyadic inference rule. Computers and Artif. Intell. 16(2), 217–227 (1997)
10. Felsenstein, J.: The number of evolutionary trees. Systematic Zoology 27, 27–33 (1978)
11. Gilks, W., Berzuini, C.: Following a moving target-Monte Carlo inference for dynamic Bayesian models. J. of the Royal Statistical Society (2001)
12. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip polynomial algorithm for sorting signed permutations by reversals. In: Proc. 27th Ann. Symp. Theory of Computing, pp. 178–189. ACM Press, New York, NY, USA (1995)
13. Huson, D., Nettles, S., Warnow, T.: Disk-covering, a fast converging method for phylogenetic tree reconstruction. J. Comput. Biol. 6(3), 369–386 (1999)
14. Larget, B., Simon, D.L., Kadane, J.B., Sweet, D.: A Bayesian analysis of metazoan mitochondrial genome arrangements. Mol. Biol. and Evol. 22(3), 486–495 (2005)
15. Liu, T., Tang, J., Moret, B.M.E.: Quartet methods for phylogeny reconstruction from gene orders. In: Gorodetsky, V., Liu, J., Skormin, V.A. (eds.) AIS-ADM 2005. LNCS (LNAI), vol. 3505, pp. 63–73. Springer, Heidelberg (2005)
16. Moret, B.M.E., Wyman, S., Bader, D.A., Warnow, T., Yan, M.: A new implementation and detailed study of breakpoint analysis. In: Proc. 6th Pacific Symp. on Biocomputing (PSB 01), pp. 583–594. World Scientific Pub, Singapore (2001)
17. Moret, B.M.E., Tang, J., Wang, L.-S., Warnow, T.: Steps toward accurate reconstructions of phylogenies from gene-order data. J. Comput. Syst. Sci. 65(3), 508–525 (2002)
18. Moret, B.M.E., Tang, J., Warnow, T.: Reconstructing phylogenies from gene-content and gene-order data. In: Gascuel, O. (ed.) Mathematics of Evolution and Phylogeny, pp. 321–352. Oxford Univ. Press, Oxford (2005)
19. Mossel, E., Vigoda, E.: Limitations of Markov Chain Monte Carlo Algorithms for Bayesian Inference of Phylogeny. Quantitative Biology, vol. 4(12) (2006)
20. Olmstead, R., Palmer, J.: Chloroplast DNA systematics: a review of methods and data analysis. Amer. J. Bot. 81, 1205–1224 (1994)

21. Palmer, J.: Chloroplast and mitochondria genome evolution in land plants. In: Herrmann, R. (ed.), Cell Organelles, pp. 99–133 (1992)
22. Pe'er, I., Shamir, R.: The median problems for breakpoints are NP-complete. Elec. Colloq. on Comput. Complexity, vol. 71 (1998)
23. Ridgeway, G., Madigan, D.: A sequential Monte Carlo method for Bayesian analysis of massive datasets. Data Mining and Knowledge Discovery 7, 301–319 (2002)
24. Raubeson, L., Jansen, R.: Chloroplast DNA evidence on the ancient evolutionary split in vascular land plants. Science 255, 1697–1699 (1992)
25. Saitou, N., Nei, M.: The neighbor-joining method: A new method for reconstructing phylogenetic trees. Mol. Biol. Evol. 4, 406–425 (1987)
26. Sankoff, D., Blanchette, M.: Multiple genome rearrangement and breakpoint phylogeny. J. Comput. Biol. 5, 555–570 (1998)
27. Swofford, D.L., Olson, G., Waddell, P., Hillis, D.M.: Phylogenetic inference. In: Hillis, D.M., Moritz, C., Mable, B. (eds.) Molecular Systematics, 2nd edn. chapter 11 (1996)
28. Tang, J., Moret, B.M.E.: Scaling up accurate phylogenetic reconstruction from gene-order data. In: Proc. 11th Conf. on Intelligent Systems for Mol. Biol. ISMB'03, in Bioinformatics, vol. 19, pp. i305–i312 (2003)
29. Tang, J.: Large-scale Phylogenetic Reconstruction from Arbitrary Gene-order Data. Ph.D. Dissertation, available online at http://www.cse.sc.edu/~jtang/dissertation.ps (2004)
30. Tang, J., Moret, B.M.E.: Linear programming for phylogenetic reconstruction based on gene rearrangements. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 406–416. Springer, Heidelberg (2005)
31. Wang, L.-S., Jansen, R., Moret, B.M.E., Raubeson, L., Warnow, T.: Fast phylogenetic methods for genome rearrangement evolution: An empirical study. In: Proc. 7th Pacific Symp. on Biocomputing (PSB 02), pp. 524–535. World Scientific Pub, Singapore (2002)

Benchmarks for Strictly Fundamental Cycle Bases*

Christian Liebchen[1], Gregor Wünsch[1], Ekkehard Köhler[2], Alexander Reich[2], and Romeo Rizzi[3]

[1] Institute of Mathematics, TU Berlin, Germany
{liebchen,wuensch}@math.tu-berlin.de
[2] Department of Mathematics, TU Cottbus, Germany
[3] Department of Mathematics and Computer Science, University of Udine, Italy

Abstract. In the MINIMUM STRICTLY FUNDAMENTAL CYCLE BASIS (MSFCB) problem one is looking for a spanning tree such that the sum of the lengths of its induced fundamental circuits is minimum.

We identify square planar grid graphs as being very challenging testbeds for the MSFCB. The best lower and upper bounds for this problem are due to Alon, Karp, Peleg, and West (1995) and to Amaldi et al. (2004).

We improve their bounds significantly, both empirically and asymptotically. Ideally, these new benchmarks will serve as a reference for the performance of any new heuristic for the MSFCB problem.

1 Introduction

Consider the following problem. Given the $N \times N$ square planar grid graph $G_{N,N}$. Find a spanning tree T such that the sum of the lengths of its induced fundamental circuits is as small as possible. Figure 1 shows a very good solution for $G_{8,8}$. Is it optimal?

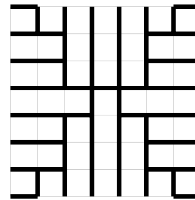


Fig. 1. A very good SFCB of $G_{8,8}$. It costs 266. Can you give a cheaper one?

At first sight, this might appear being a kind of "toy problem." Indeed, at the occasion of its annual web-based Christmas quiz (www.mathe-kalender.de), on December 18, 2006 the DFG Research Center MATHEON essentially asked the above question to more than 9 000 registered users (pupils, teachers, scientists, and others). Typically, each day about 1 500 users post their answers, and more than 60% of these answers are correct. In contrast, on Dec. 18, less than 15% of the answers were correct. What makes this problem so hard?

Given a spanning tree T in a graph G; the fundamental circuits with respect to T form a strictly fundamental cycle basis (see Section 2 for formal definitions). We refer to the problem of finding a spanning tree whose fundamental circuits sum to a minimum

* Supported by the DFG Research Center MATHEON in Berlin and by the DFG Research Training Group GK-621 "Stochastic Modelling and Quantitative Analysis of Complex Systems in Engineering" (MAGSI).

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 365–378, 2007.
© Springer-Verlag Berlin Heidelberg 2007

value as the MINIMUM STRICTLY FUNDAMENTAL CYCLE BASIS (MSFCB) Problem. As a generalization, in the MINIMUM CYCLE BASIS (MCB) Problem one seeks for a general cycle basis of minimum length.

Applications. There is a variety of applications for the MCB problem. These include biology and chemistry ([10]), traffic light planning ([14]), periodic railway timetabling ([16]), and electrical engineering ([6]). Typically, cycle bases are computed during a preprocessing phase. During the actual computations one ensures that a certain problem-specific property is true for the elements of the selected cycle basis in the graph of interest. By the properties of cycle bases, one can conclude that this particular property is actually true for *any* cycle in the graph, right as it is required by the practical application. In many cases it can be observed that shorter cycle bases imply a shorter time for the actual computations.

For some of these applications not all cycle bases are of use (e.g. traffic light planning and periodic railway timetabling); however, strictly fundamental cycle bases—being the most specialized ones—always are. In other applications, such as electrical engineering, it is at least much more favorable to use strictly fundamental cycle bases, because of the numerical stability of the subsequent calculations ([3]). The practical relevance of the MSFCB problem is reflected by numerous computational studies by different groups working in combinatorial optimization ([2,7,8,11,15,20]). We give a short overview of their results in the following.

Theory. Already in 1982 Deo et al. ([7]) proved the MSFCB problem to be NP-hard for general unweighted graphs. Because of the practical relevance of those cycle bases for various applications, many heuristics were proposed and tested. However, for none of these heuristics any non-trivial approximation ratio or any non-trivial bound on the absolute length of the resulting bases was shown. The only statement going into that direction is a conjecture by Deo et al. ([7]) predicting that MSFCBs of unweighted graphs have length $O(n^2)$.

The design of most of these heuristics has been led by the following observation: "A BFS produces spanning trees of short diameters. Thus, the BFS method on the average generates fundamental cycles of shorter total length (compared to some other approaches)." ([7]). In particular, these heuristics make local decisions that are mainly based on the degrees of the vertices, either in G or in some residual graph.

Elkin et al. ([9]) use some completely different technique. They consider the average-stretch tree spanner problem. Profiting from the Unified Notation for Tree Spanner problems (UNTS, [17]) one can conclude that in the case of unweighted graphs their results can be applied immediately to the MSFCB problem. In particular, their recursive algorithm computes a SFCB of asymptotic length $O(m \cdot \log^2 n \log\log n)$. Note that this is the first non-trivial theoretical guarantee on the quality of a solution to the SFCB problem, and it is obtained by a recursive approach. Moreover, for graphs with $|E| \in O(n^2/\log^2 n \log\log n)$, this result proves Deo's conjecture.

Why Planar Grids? Due to the absence of theoretical bounds for the degree-based heuristics, the authors of these approaches used empirical calculations to evaluate the quality of their algorithms. Yet, to compare different heuristics empirically, it is essential to run them on the very same input graphs. But what are *good* testbeds?

Liberti et al. ([15]) consider square planar grid graphs being "the most difficult testbeds for the MSFCB problem, both for heuristic and exact methods, due to the huge quantity of configurations having the same SFCB cost." In fact, also from a theoretical point of view this can be motivated in three different ways. First, these graphs are almost regular because more than $n-4\sqrt{n}$ vertices have degree four—a nightmare for any degree-based heuristic. Second, within a fixed distance, the subgraphs around almost each vertex are isomorphic. Hence, any heuristic that bases its decisions on local configurations is likely to perform poorly. Third, if G was a tree, then in the MSFCB problem no decisions are to be made and the problem clearly becomes trivial. An appropriate measure for the tree-alikeness of a graph is its tree-width ([4]). And with respect to that measure, grid graphs—having $\Theta(n)$ edges and tree-width $\sqrt{n}$—are prominent examples of being far away from being a tree ([21]). Thus the MSFCB problem is likely to keep its hardness when considered on planar grids. Finally, also from a practical perspective planar grids are very suitable. Many of the relevant instances in several applications are planar graphs or even planar grids (e.g. electrical engineering, traffic light scheduling).

Focusing on grid graphs could appear narrow. But it is commonly believed that these hold the key to better algorithms for cycle bases. Indeed, for square planar grid graphs Alon, Karp, Peleg, and West ([1]) design spanning trees that induce cycle bases of length $\frac{4}{3}n\log_2 n+O(n)$ ([13]). They prove that these trees are asymptotically optimal. Moreover, they conjecture their trees are "essentially optimal." Using this asymptotical upper bound we can demonstrate how degree-based heuristics may fail. The degree-based $\mathcal{C}$-order heuristic ([15]) can be implemented to compute "Machete"-trees (cf. [5], and Figure 1 for an example). These trees do not only minimize the diameter of trees in grids, but also the maximum stretch. At first sight these two parameters of a tree could appear being tightly related to its associated SFCB cost. It is again the UNTS ([17]) which makes it transparent that even for unweighted graphs no two of these three measures do actually coincide. Machete-trees yield MSFCB objective values of $\Theta(n^{\frac{3}{2}})$. Thus degree-based heuristics risk to fail drastically on grid graphs. This again underlines the property of grids being a relevant testbed.

In fact, Liberti et al. ([15]) also select grid graphs as one of their testbeds. On the 50×50 grid they observe that their new $\mathcal{C}$-order heuristic attains an objective value of 46 452, compared to 48 254 of the NT heuristic ([8]). Unfortunately, this isolated comparison does not clarify whether these are indeed good objective values. Amaldi et al. ([2]) consider grid graphs in their computations as well. They report a solution of objective value 23 026, that was obtained by local search techniques. This clearly shows the need for good benchmark values for the MSFCB problem for the particularly challenging case of planar grid graphs—also for the future evaluation of new heuristics.

Of course, relevant benchmarks also include dual bounds. Since general cycle bases are a superset of strictly fundamental cycle bases, the value of an MCB clearly serves as a lower bound for the value of an MSFCB. On grid graphs, this yields a lower bound of $4\cdot(\sqrt{n}-1)^2$. Yet, exploiting the particular structure of grid graphs one can achieve lower bounds for the MSFCB problem that are (asymptotically) even better. The first was given in [1] and it has value $\frac{\ln 2}{2048}n\log_2 n-O(n)$.

Contribution. The above discussion motivates the need for a collection of benchmark values for the MSFCB problem on square planar grid graphs. We suggest two new families of lower bounds and two new families of upper bounds.

In Section 3 we sketch a proof of Köhler et al. ([13]) on how a new approach improves the asymptotic lower bound by Alon, Karp, Peleg, and West ([1]) to $\frac{1}{12}n\log_2 n - O(n)$, i.e., by a factor of more than 245. In addition, we prove $6n - 20\sqrt{n} + 22$ to be a new lower bound—for $N \in \{3,\dots,61\}$ this constitutes the best lower bound known. It is a fact that *all* the primal solutions (upper bounds) that so far have been proposed in the literature are of grids with dimension within this range.

Finally, in Section 4 we introduce a new scheme for constructing very short strictly fundamental cycle bases—both empirically and asymptotically. We prove an upper bound on the length of their SFCB of $0.97n\log_2 n + O(n)$, hereby improving the objective value $\frac{4}{3}n\log_2 n + O(n)$ of the spanning trees due to Alon, Karp, Peleg, and West ([1]), which they assumed being essentially optimal. In our experiments we also compare the lengths of those trees to spanning trees that were obtained using local search techniques ([2]). Our new trees improve the best known solutions for all $N \geq 20$. Interestingly, for $N = 10, 15, \dots, 55$ these solutions are even local optima.

2 Preliminaries

We consider 2-connected simple undirected unweighted[1] graphs $G = (V,E)$ with $n = |V|$ and $m = |E|$. $\nu = m - n + 1$ denotes the *cyclomatic number* of G. Let C be a circuit (cf. [22, Ch. 3]) in G and denote by γ_C its $\{0,1\}$-incidence vector. The *cycle space* $\mathcal{C}$ of G is the following vector subspace over GF(2),

$$\mathcal{C} := \operatorname{span}\left(\{\gamma_C \,|\, C \text{ circuit in } G\}\right).$$

A *cycle basis* B of G is a set of ν circuits of G whose incidence vectors are a basis of $\mathcal{C}$. The *length* $\Phi(B)$ of a cycle basis of an unweighted graph is defined as $\Phi(B) = \sum_{C \in B} |C|$. A *minimum cycle basis* (MCB) of a graph G is a cycle basis of G of minimum length.

Now let T be some spanning tree of G. Depending on the context, we either regard T as a subgraph of G or as a set of edges $T \subset E$.

For $e \in E \setminus T$, we denote by $C_T(e)$—or C_e for short—the *fundamental circuit* that e induces with respect to T, i.e., the unique circuit in $T \cup \{e\}$. There are ν fundamental circuits associated to T. These form a cycle basis which is called *strictly fundamental* (here, we may write $\Phi(T)$ instead of $\Phi(B)$). Consequently, a *minimum strictly fundamental cycle basis* (MSFCB) is a strictly fundamental cycle bases having minimum length. In the context of local search algorithms we make use of some neighborhood concept. For an arbitrary spanning tree T we define its *neighborhood* as the set of spanning trees T' such that $|T \cap T'| \geq |T| - 2$.

For $N \in \mathbb{N}$ we define the planar *grid graph* $G_{N,N}$ in the usual way as the graph on $V = \{1,\dots,N\} \times \{1,\dots,N\}$ with

$$E = \{\{(i,j),(i',j')\} : |i-i'| + |j-j'| = 1\} = \{\{u,v\} : ||u-v||_1 = 1\}.$$

[1] Of course, minimum cycle basis problems are also investigated for weighted graphs. But as we aim to contribute to the particularly challenging case of planar unweighted grid graphs, we omit edge weights throughout our presentation.

In a graphical representation, e.g., in an embedding into $\mathbb{Z}^2$, the first index of a vertex represents its x-coordinate, the second index its y-coordinate. Obviously, $G_{N,N}$ contains $n = N^2$ vertices and $m = 2 \cdot N \cdot (N-1)$ edges; its cyclomatic number ν is $(N-1)^2$.

The dual of an embedded planar graph G is denoted by G^*. Hence, $(G_{N,N})^*$ is again the graph of a $(N-1) \times (N-1)$ grid plus a further vertex F^∞, which corresponds to the outer face of the initial embedded planar graph. Recall that the edge set of G can be identified with the edge set of G^* (see [22, Ch. 3]). Now, consider a spanning tree T of $G_{N,N}$ and its dual counterpart, that we denote by T^*. In fact, T^* can be understood as the complement of T, in the sense that it contains for each edge in $E(G_{N,N}) \setminus T$ the corresponding counterpart from G^*. Then, T^* is a spanning tree of G^*, although it is not necessarily connected when restricted to $G^* \setminus \{F^\infty\}$.

3 New Lower Bounds

A trivial lower bound for the MSFCB problem is the length of a minimum cycle basis. It can be computed using polynomial-time algorithms (e.g. [12]). However, by a result of Alon et al. ([1]) these lower bounds may miss the optimum value of the MSFCB problem by a logarithmic-factor. In particular this is true for square planar grid graphs $G_{N,N}$. Here, the trivial lower bound is only $4 \cdot (\sqrt{n}-1)^2$ whereas Alon et al. proved $\Phi(T) \geq \frac{\ln n}{2048} n - O(n)$ for all spanning trees T.

In Corollary 1 we identify $6n - 20\sqrt{n} + 22$ as a lower bound for the MSFCB problem on a square planar grid. This is obtained using purely combinatorial arguments. Alternatively, mixed-integer linear programs (MIP) could yield lower bounds. However, even sophisticated MIP formulations ([2,15]) in combination with several valid inequalities ([19]) do not improve the combinatorial lower bound.

On the one hand, there is some dimension N_0 such that our lower bound gets dominated by the asymptotically better lower bound by Alon et al. ([1]). On the other hand, in the case of $N = 2^k + 2$, $k \in \mathbb{N}$, we present one further lower bound function with

$$\Phi(T) \geq \frac{1}{12} n \log_2 n - O(n), \tag{1}$$

being due to Köhler et al. ([13]) and which dominates Alon et al.'s lower bound. Furthermore, it illustrates the predominance of the $6n - 20\sqrt{n} + 22$ bound over the trivial lower bound: The function that interpolates the asymptotically best lower bound (1) intersects with the trivial lower bound already for $N_1 \approx 8.1$. In contrast, $6n - 20\sqrt{n} + 22$ intersects (1) not before $N_2 \approx 61.6$; this corresponds to a graph with more than 7 300 edges.

The remainder of this section is—to that extent—subdivided into two parts. First we examine small grids and sketch the best known bound here: $6n - 20\sqrt{n} + 22$. Afterwards, we outline the proof of $\frac{1}{12} n \log_2 n - O(n)$ being a lower bound which works out only on very large grids. For detailed proofs see [19].

Small Grids. In this paragraph we now sketch how for small dimensional grids the lower bound mentioned above can be derived. Let C be some circuit in $G_{N,N}$. We denote by $\mathrm{diam}_H(C)$ the *horizontal diameter* of C, i.e., the difference between the minimum and the maximum x-coordinates of vertices in C in the $\mathbb{Z}^2$ embedding. Similarly, we define the *vertical diameter* of C and denote it by $\mathrm{diam}_V(C)$. In particular,

$$|C| \geq 2 \cdot (\text{diam}_H(C) + \text{diam}_V(C)). \tag{2}$$

For $e \notin T$, we use $\text{diam}_H(e) := \text{diam}_H(C_T(e))$ as a short hand.

Let C be a circuit in $G_{N,N}$ and consider its enclosed finite region R. We define $\mathring{C}$ to be the set that collects all the edges of $E(G_{N,N})$ that are incident with two faces of $G_{N,N}$ that have empty intersection with $\mathbb{R}^2 \setminus R$. In other words, $\mathring{C}$ refers to the edges *inside* C.

Proposition 1. *Let $G_{M,N}$ be the $M \times N$ planar grid, let T be a spanning tree of it, and let C be a simple circuit in $G_{M,N}$. Then*

$$\sum_{e \in (C \cup \mathring{C}) \setminus T} |C_T(e)| \geq 4 \cdot |C \setminus T| + 6 \cdot |\mathring{C} \setminus T|. \tag{3}$$

Proof. Using (2) it suffices to show that

$$\sum_{e \in (C \cup \mathring{C}) \setminus T} 2 \cdot (\text{diam}_H(e) + \text{diam}_V(e)) \geq 4 \cdot |C \setminus T| + 6 \cdot |\mathring{C} \setminus T|. \tag{4}$$

We derive a lower bound for $\sum_{e \in (C \cup \mathring{C}) \setminus T} \text{diam}_H(e) + \text{diam}_V(e)$ by defining a function $d(e)$ such that

$$\text{diam}_H(e) + \text{diam}_V(e) \geq d(e), \quad \text{for all } e \in (C \cup \mathring{C}) \setminus T. \tag{5}$$

We define the function $d(e)$ as follows. Since e is not contained in T we already know that

$$\text{diam}_H(e) \geq 1 \quad \text{and} \quad \text{diam}_V(e) \geq 1. \tag{6}$$

To increase $d(e)$ beyond two, consider the spanning tree T^* in the dual graph $(G_{N,N})^*$ that corresponds to $E(G_{N,N}) \setminus T$. Take F^∞ as the root of T^*. Consider the two faces of $G_{N,N}$ that are incident with e. We refer to the one with larger distance from F^∞ in T^* as $F(e)$.

For each edge $f \in (F(e) \setminus (C \cup \{e\})$, denote by $F(f) \neq F(e)$ the other face that f is incident with. Observe that $F(f) \neq F^\infty$ because of $f \notin C$. By the grid structure, each of these faces $F(f)$ is in a different direction with respect to $F(e)$, i.e. either north, east, south, or west. Now, if $f \notin T$, we know that $C_T(e)$ also has to contain $F(f)$ in its enclosed bounded region. This way, such an edge $f \in (F(e) \setminus (C \cup T \cup \{e\}))$ serves as a certificate that any lower bound on either $\text{diam}_H(e)$ or $\text{diam}_V(e)$, respectively, can be incremented. In total, we set

$$\text{diam}_H(e) + \text{diam}_V(e) \geq 2 + |F(e) \setminus (C \cup T \cup \{e\})| =: d(e), \tag{7}$$

which guarantees that (5) is still true.

When summing over all edges $e \in C \cup \mathring{C}$, we may rearrange the summation. To this end, observe that each edge $f \in \mathring{C} \setminus T$ has precisely one dual parent e among $(C \cup \mathring{C}) \setminus T$. Hence, it increments the lower bound on $\text{diam}_H(e) + \text{diam}_V(e)$ for precisely this one edge e. In other words, each edge $f \in \mathring{C} \setminus T$ counts three times: according to (6) it counts $d(f) = 2$ for its proper fundamental circuit $C_T(f)$, *plus* one for precisely its unique parent edge $e \in (C \cup \mathring{C}) \setminus T$. To summarize,

$$\sum_{e \in (C \cup \mathring{C}) \setminus T} d(e) \overset{(7)}{=} 2 \cdot |C \setminus T| + 3 \cdot |\mathring{C} \setminus T|. \tag{8}$$

Finally, we conclude that

$$\sum_{e \in (C \cup \mathring{C}) \setminus T} |C_T(e)| \overset{(2)}{\geq} \sum_{e \in (C \cup \mathring{C}) \setminus T} 2 \cdot (\mathrm{diam}_H(e) + \mathrm{diam}_V(e))$$
$$\overset{(5)}{\geq} \sum_{e \in (C \cup \mathring{C}) \setminus T} 2 \cdot d(e) \overset{(8)}{\geq} 4 \cdot |C \setminus T| + 6 \cdot |\mathring{C} \setminus T|. \qquad \square$$

Corollary 1. *Let $N \geq 3$ and $G_{N,N}$ be the $N \times N$ planar grid with $n = N^2$ vertices. Then for each spanning tree $T \subset E$*

$$\Phi(T) = \sum_{e \in E \setminus T} |C_T(e)| \geq 6 \cdot n - 20\sqrt{n} + 22. \tag{9}$$

Proof. Simply take C as the circuit that contains precisely the edges that are incident with F^∞. Because of $E = C \dot\cup \mathring{C}$ we apply Proposition 1 to C. There, we minimize the RHS in (3) by maximizing $|C \setminus T|$. Now consider the four vertices which are not incident to any edge in $\mathring{C}$. In any tree T, these vertices must be incident with one edge in $C \cap T$. As $N \geq 3$, we conclude that $|C \cap T| \geq 4$, thus $|C \setminus T| \leq 4\sqrt{n} - 8$. Finally, a simple calculation yields (9). $\square$

Observe that $6n - 20\sqrt{n} + 22 \geq 4 \cdot n - 8\sqrt{n} + 4$, for all $N \geq 3$ and $n = N^2$.

As a special case, consider a spanning tree T with the following property: each edge $e \in E \setminus T$ has distance at most two to F^∞ in T^*. This implies that (7) holds with equality. As a consequence, one can argue that also in (9) we will find equality. Note that for $N \in \{3,4,5\}$ such spanning trees do actually exist. Hence, it follows that in these dimensions the bound of Corollary (1) is the optimum value of the MSFCB problem.

Large Grids. We sketch the following result: The strictly fundamental cycle basis B of any spanning tree T in the square $N \times N$ grid with $n = N^2 = (2^k + 2)^2$ vertices satisfies $\Phi(T) \geq \frac{1}{12} n \log_2 n - O(n)$. This direct approach substantially improves the lower bound that has been obtained by Alon, Karp, Peleg, and West in [1, Thm. 6.6] by a factor of more than 245.[2] Due to space limitations, we cannot present any of our new proofs here. Instead we refer to [13] for the complete analysis.

In contrast to [1] we decided to tackle the lower bound problem from the perspective of the dual planar graph G^*. In particular, we make use of the fact that for each spanning tree T, there is a one-to-one correspondence between its induced fundamental circuits in G, and its induced fundamental cuts in G^*. More precisely, if an edge $e \in E \setminus T$ induces a circuit in G, then its dual counterpart induces a cut in G^*—and both contain the very same edges.

To detect sufficiently long circuits, according to inequality (2) we concentrate on circuits that have large horizontal and/or vertical diameters. To obtain the claimed lower bound, it is even sufficient to consider only the horizontal diameter $\mathrm{diam}_H(C)$ or only the vertical diameter $\mathrm{diam}_V(C)$ of a circuit C.

[2] We point out that the authors of [1] state explicitly that they were not trying to "optimize constants."

Now, for each such level-vertex v, we consider the unique subpath P of T^* that connects v with F^∞. We follow this path from v up to the first edge e that is incident with the border of B_u. Assume, w.l.o.g., that this edge e is "north" of v. In the primal grid, e is, in fact, a horizontal edge. Then we only consider the subsequence of vertical edges P_V (all being horizontal edges in $G_{N,N}$) of $P \subseteq T^*$ such that each edge is by one closer to e than its predecessor in P_V. In [13] we call this subpath the vertical *pseudo-path* of P. Then, for a vertex v of one particular level ℓ, $\ell \in \{1,\ldots,k\}$, we know that the $2^{\ell-1}$ edges of P_V induce fundamental circuits with respect to T of vertical diameters at least $1,2,\ldots,2^{\ell-1}$.

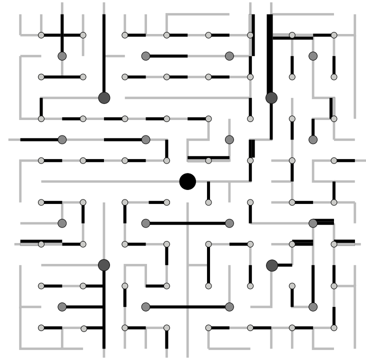


Fig. 2. The dual tree T^* of a spanning tree T in $G_{18,18}$. In our lower bound for $\Phi(T)$ we only sum lower bounds on the fundamental circuits that are induced by the black edges.

Now, we aim at summing the lower bounds on the diameters of the fundamental circuits for each occurrence of an edge on some pseudo-path. It is a simple observation that only pseudo-paths of different levels could share a, say, horizontal edge e of $G_{N,N}$, and thus potentially cause some double-counting (Lem. 3 in [13]). But one can further observe that the lower bound on $\text{diam}_V(e)$ or $\text{diam}_H(e)$, respectively, is always larger for the pseudo-path that we defined for the vertex with the higher level index. Hence, to prevent us from any double-counting, for such an edge e we count as lower bound on $|C_T(e)|$ only the bound on $\text{diam}_V(e)$ that we identify on the highest level. As a consequence we get the following theorem.

Theorem 1 ([13]). *Let $G_{N,N}$ be the planar grid graph with $n = N^2 = (2^k+2)^2$ vertices. For every spanning tree T of $G_{N,N}$ there holds*

$$\Phi(T) \geq \frac{1}{12} n \log_2 n - O(n). \tag{10}$$

4 New Upper Bounds

Alon et al. ([1]) provided spanning trees T_{AKPW} and showed that the induced strictly fundamental cycle bases are bounded from above by $2n\log_2 n + o(n\log_2 n)$. An exact counting ([13]) even revealed that

$$\Phi(T_{\text{AKPW}}) \leq \frac{4}{3} n \log_2 n + O(n), \tag{11}$$

where $N = \sqrt{n}$, and $N = 2^k$ for some $k \in \mathbb{N}$. Although Alon, Karp, Peleg, and West ([1]) considered their trees to be "essentially optimal", we can construct trees with an asymptotic coefficient for the $n\log_2 n$ term being strictly smaller than one. Moreover, we present trees with very good empirical performance already in small dimensions.

For that we introduce a class of recursively defined trees which are the union of spanning trees of certain rectangular subgraphs of $G_{N,N}$. More precisely, we form a

partition of $G_{N,N}$ in rectangular subgraphs, each having a certain aspect ratio $\alpha \geq 1$. For our construction we now specify certain values of α and show how to construct a spanning tree for some grid $G_{M,N}$ having *aspect ratio* $\max\{\frac{M}{N}, \frac{N}{M}\} \approx \alpha$. This is done recursively. Assume, w.l.o.g., that $M \geq N$. At the top-level of the recursion, we add to $T_\alpha(G_{M,N})$ all the edges of the two longer borders of $G_{M,N}$ (here the $2M$ horizontal border edges), plus the edges of one of its two other borders (see the left picture in Figure 3). For the recursion, we partition the faces of $G_{M,N}$ into almost equally-sized rectangular

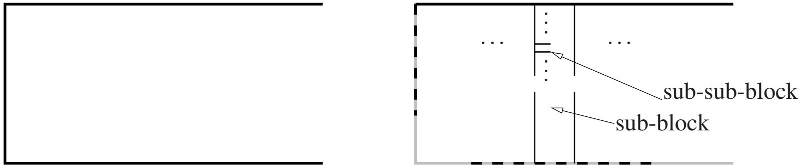


Fig. 3. The shape of a block (left) and with a sketched interior recursively filled with smaller blocks (right), always keeping the aspect ratio

subgraphs of aspect ratio again being close to α. Only the faces of one horizontal path in $(G_{M,N})^*$, located almost in the middle of its two horizontal borders, are not contained in any of these rectangular subgraphs.

Note that these trees are closely related also to other families of trees. In $G_{N,N}$, choosing $\alpha \geq \frac{N/2}{1}$ there exists a partition of the grid such that we end up with Machete-trees ([5], or see Figure 1). An aspect ratio of $\alpha = 1$ yields trees which can be obtained alternatively by a construction similar to the one for T_{AKPW}.

In what follows we will structure our presentation into two parts, one part considering large grids and a second part dealing with small grids. The section on large grids shows how to achieve good asymptotic bounds whereas the section on small grids provides better empirical results. For the two types of grids (small and large) we will introduce trees with a block structure having two different aspect ratios. In the one case we will use an aspect ratio of approximately 3 : 1, in the other case an aspect ratio of 2 : 1. In addition, the two cases differ in how the blocks are actually used to define a tree. Whereas on large grids it is sufficient to *cover* the grid with three (almost) equally-sized 3 : 1 blocks, for small dimensions the grids are *tiled* with a large number of 2 : 1 blocks of many different sizes.

Large Grids. To achieve a good asymptotical upper bound we decided to construct trees using the above described blocks with an aspect ratio of 3 : 1. Since it is not obvious how to subdivide or tile a square grid of arbitrary dimension with these particular blocks, we construct our trees in a bottom-up fashion. More precisely, we take an atomic block of dimension 6×14 and arrange 32 copies of such a block to a new one having dimension 81×28. This procedure is then iterated providing spanning trees for dimensions

$$\left(\frac{78}{31} \cdot 32^{k/2} + \frac{15}{31}\right) \times \left(\frac{419}{496} \cdot 32^{k/2} + \frac{30}{31}\right) \tag{12}$$

with k chosen integral and even (see [19] for details). Finally, three copies of such a tree can be put together and cover the entire square grid.

For a discussion on bounding the small overhanging part when following this stacking as well as for the precise analysis we again refer to [19]. As a result we identify the following asymptotically upper bound.

Lemma 1 ([19]). *Let $G_{N,N}$ denote the $N \times N$ square planar grid with $n = N^2$ vertices and with $N = \frac{78}{31} \cdot 32^{k/2} + \frac{15}{31}$ for some even integer k. Then the size of a minimum strictly fundamental cycle basis on $G_{N,N}$ can be bounded from above by*

$$0.978 \cdot n \cdot \log_2 n + O(n).$$

Small Grids. The $3:1-$block structured trees, as described in the above paragraph are not perfectly suited for smaller dimensions. As shown, $3:1$ is asymptotically a very good aspect ratio. Yet, it is not possible to decompose an arbitrary square grid into $3:1-$blocks without losing much of their advantage because of "rounding errors."

Therefore, for small grids, we chose a different block-structured graph. This time we use an aspect ratio of $2:1$. In contrast to the above, the $2:1-$blocks, do not really cover, but rather tile the square grid. The tiling procedure roughly goes as follows:

At first, two opposite $2:1-$blocks are put in the middle of the grid. See for example the two blocks marked with "A", having side lengths 8×15 in Figure 4. Next, horizontal $2:1-$blocks (marked with "B") are added centrally aside such that rectangular subgrids in the four corners remain. In those corners (marked "C") we always direct the next block such that its depth can be chosen as small as possible, while its aspect ratio should stay as close as possible to the target ratio $2:1$. During this procedure we do not need to pay attention to any rounding inaccuracies. In Figure 4 an example $2:1-$block structured tree for dimension $N = 31$ is shown.

The empirical quality of the so defined trees for small grids will be evaluated in the next section.

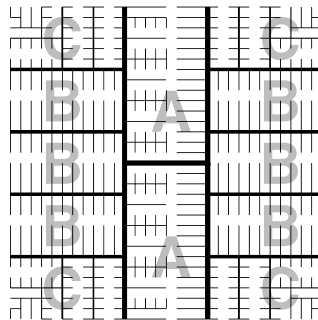


Fig. 4. Notice the parquet-like structure of the tree with tiles having height-width ratio of 2 with small errors due to roundings. Inside, the blocks themselves are recursively filled with smaller blocks still maintaining the $2:1$ ratio.

5 Experimental Results

In this section we compare different spanning trees with respect to the length of the strictly fundamental cycle basis they induce. In addition to the degree-based tree-growing heuristics that we already referred to in the introduction, local search techniques have been considered, too.

Amaldi et al. ([2]) reported the performance of several strategies for searching the neighborhood of a spanning tree. In what they denote by local search (LS), the entire neighborhood is examined and they move to the tree with the best improvement. In a second deterministic strategy (ES, for local search with edge sampling) only a restricted number of neighbors are tested.

To prevent LS to terminate too early in a too bad local optimum, Amaldi et al. ([2]) run metaheuristics such as variable-neighborhood search (VNS) and a tabu search (TS) on top of LS. In any of their computations, an adapted version of the tree-growing heuristic of [20] is used as the initial solution.

In our computations, we use the $2:1-$block-structured tree as initial solution. In contrast to (LS) we do not examine the entire neighborhood for improvement. Instead, whenever we identify a neighbor that improves the current solution, we greedily move to that neighbor. Of course, this method depends on the order in which the edges in the tree are checked. Empirical studies showed, however, that the influence of the edge-order is neglectable. For our computational studies we chose a random order of edges and ran our greedy-like approach—denoted by (GS)—ten times, considering the best value of the length of the cycle basis and the according running time of (GS). We skip average values, because we see the goal of the study in giving benchmark results. Examining the quality of the heuristic is only a secondary goal. Among the ten sample runs the lengths of the cycle bases vary by less than 1% only, anyway.

Results. In Table 1 we compare the constructive heuristics, i.e., those that build up a tree without doing any subsequent local improvements. Moreover, we complement these values with information on lower bounds obtained by Corollary 1 and a minimum cycle basis. The latter were also used in the recent study of Amaldi et al. ([2]). Note further, that $N = 130$ is the dimension closest to 100 for which our asymptotic bound is defined exactly. We mention that for this value of N the bound that we derived in Corollary 1 is only about 3.5% weaker than our asymptotic bound.

In our tables the *italic* numbers highlight the best known upper and lower bounds. For $N = 5$, these coincide and we mark this in boldface. Observe that for any dimension $N \geq 10$, the new trees that we propose in Section 4 yield smaller SFCB values than any of the other constructive heuristics.

In Table 2 we compare the different local-search-type heuristics. For our greedy search (GS) we used a 3.2GHz Intel P4 computer ("A1"), running Linux and using LEDA©. Amaldi et al. used for their local search heuristics (LS) and (ES) also an Intel P4 computer running Linux, with 2.66GHz ("A2"). Accordingly, the times stated in Table 2 refer to the particular architecture. The values for the meta-heuristics (TS) and (VNS)—also quoted from [2]—each refer to 10 minute runs on the A2 environment. Similarly as in the purely constructive context, our new solutions improve the best known upper bounds for all dimensions $N \geq 20$.

Table 1. Comparison of the cost of some selected trees, i.e., the length of the according strictly fundamental cycle bases. The rightmost column presents the previously best known lower bound for small dimensions, obtained just by $4 \cdot (N-1)^2$. The penultimate column now states the consistently better lower bounds due to Corollary 1.

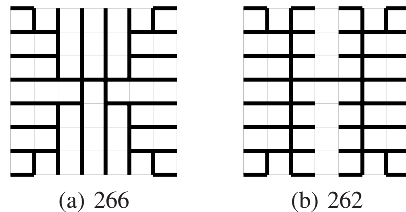
N	"2 : 1"	AKPW [1]	Machete [5]	C-Order [15]	Deo's NT [8,15]	Corollary 1	MCB
5	76	78	**72**	**72**	78	**72**	64
10	468	524	492	492	518	*422*	324
15	1 300	1 554	1 512	1 512	1 588	*1 072*	784
20	2 550	3 030	3 382	3 382	3 636	*2 022*	1 444
25	*4 368*	5 410	6 352	6 352	6 452	*3 272*	2 304
30	*6 656*	8 408	10 672	10 672	11 638	*4 822*	3 364
35	*9 592*	11 694	16 592	16 592	16 776	*6 672*	4 624
40	*13 162*	16 078	24 362	24 362	28 100	*8 822*	6 084
45	*17 236*	21 784	34 232	34 232	35 744	*11 272*	7 744
50	*21 920*	27 912	46 452	46 452	48 254	*14 022*	9 604
55	*27 356*	35 124	61 272	61 272	62 026	*17 072*	11 664
60	33 406	42 790	78 942	78 942	92 978	*20 422*	13 924
70	*47 300*	59 244	123 832	–	–	*28 022*	19 044
80	63 964	80 678	183 122	–	–	*36 822*	24 964
90	83 412	108 012	258 812	–	–	*46 822*	31 684
100	106 090	137 390	352 902	–	–	*58 022*	39 204

Table 2. An overview of the quality of five local search approaches. Missing values are marked with an "–" and running times are measured in *mm*:*ss*. The columns (LS)–(TS) are cited from [2].

N	(GS) cost	(GS) time	(LS) cost	(LS) time	(ES) cost	(ES) time	(VNS) cost	(TS) cost
5	**72**	00:00	**72**	00:00	74	00:00	**72**	**72**
10	468	00:00	474	00:00	524	00:00	*466*	*466*
15	1 300	00:00	1 318	00:00	1 430	00:00	1 280	*1276*
20	2 550	00:00	2 608	00:03	3 186	00:00	2 572	2590
25	*4 368*	00:00	4 592	00:16	5 152	00:02	4 464	4430
30	*6 656*	00:01	6 956	00:47	8 488	00:03	6 900	6882
35	*9 592*	00:02	10 012	02:19	11 662	00:08	9 982	9964
40	*13 162*	00:07	13 548	06:34	15 924	00:26	13 524	13534
45	*17 236*	00:06	18 100	14:22	22 602	01:00	18 100	18100
50	*21 920*	00:09	23 026	31:04	33 274	01:10	23 026	23552
55	*27 340*	00:31	–	–	–	–	–	–
60	*33 374*	01:01	–	–	–	–	–	–
80	*63 810*	07:24	–	–	–	–	–	–
90	*83 222*	07:48	–	–	–	–	–	–
100	*105 766*	14:01	–	–	–	–	–	–

As already mentioned before we ran our local search (GS) with a random order of the edges. In Table 2 the first two columns present the value for the best run out of ten samples, and the according running time, respectively.

However, it has to be mentioned that only for dimensions $N \in \{60, 80, 90, 100\}$ the start tree had *not* already been locally optimal.



(a) 266 (b) 262

Fig. 5. Did you find a better tree of cost only 262? Indeed, the tree on the right hand side is an optimum solution for $G_{8,8}$

6 Conclusions

A summary of this paper has to be twofold. On the one hand, on square planar grid graphs—being a particularly challenging family of graphs for the MSFCB Problem—we significantly improved the lower and upper bounds that were previously known for this problem.

On the other hand, optimality proofs remain non-trivial. For instance, from dimension $N \geq 6$ on, the lower bound obtained in Corollary 1 is not tight. In particular it cannot prove the optimality of the SFCB in Figure 5(b). We are only able to achieve optimality proofs for $6 \leq N \leq 10$ by using combinatorial arguments that extend the ones used in the proof of Proposition 1 ([18]). Nevertheless, further efforts are to be made in order to tackle larger dimensions.

Furthermore, columns 4–6 of Table 1 illustrate to what extent degree-based heuristics for the MSFCB problem are inferior to applying recursive approaches. In other words, for any heuristic for the MSFCB problem we recommend to evaluate it on planar square grid graphs and compare its performance to the values that we provide in Tables 1 and 2.

Acknowledgment

We thank Janina Brenner for intensive discussions on the empirical lower bounds.

References

1. Alon, N., Karp, R.M., Peleg, D., West, D.B.: A graph-theoretic game and its application to the k-server problem. SIAM J. Comput. 24(1), 78–100 (1995)
2. Amaldi, E., Liberti, L., Maculan, N., Maffioli, F.: Efficient edge-swapping heuristics for finding minimum fundamental cycle bases. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 14–29. Springer, Heidelberg (2004)
3. Bächle, S., Ebert, F.: Graph theoretical algorithms for index reduction in circuit simulation. Preprint 245, DFG Research Center Matheon (2005)
4. Bodlaender, H.L.: A tourist guide through treewidth. Acta Cybern. 11(1-2), 1–22 (1993)
5. Boksberger, P.: Minimum stretch spanning trees. Diploma thesis, ETH Zürich (2003)
6. Bollobás, B.: Modern Graph Theory. In: Graduate Texts in Mathematics, 2nd printing vol. 184, Springer, Heidelberg (2002)

7. Deo, N., Krishnomoorthy, M., Prabhu, G.: Algorithms for generating fundamental cycles in a graph. ACM Transactions on Mathematical Software 8(1), 26–42 (1982)
8. Deo, N., Kumar, N., Parsons, J.: Minimum-length fundamental cycle set: New heuristics and an empirical study. Congressus Numerantium 107, 141–154 (1995)
9. Elkin, M., Emek, Y., Spielman, D.A., Teng, S.-H.: Lower-stretch spanning trees. In: Gabow, H.N., Fagin, R. (eds.) STOC, pp. 494–503. ACM Press, New York (2005)
10. Gleiss, P.M.: Short Cycles. Ph.D. thesis, Universität Wien (2001)
11. Gotlieb, C.C., Corneil, D.G.: Algorithms for finding a fundamental set of cycles for an undirected linear graph. Communications of the ACM 10(12), 780–783 (1967)
12. Kavitha, T., Mehlhorn, K., Michail, D., Paluch, K.E.: A faster algorithm for minimum cycle basis of graphs. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 846–857. Springer, Heidelberg (2004)
13. Köhler, E., Liebchen, C., Rizzi, R., Wünsch, G.: Reducing the optimality gap of strictly fundamental cycle bases in planar grids. Preprint 007/2006, TU Berlin, Mathematical Institute (2006)
14. Köhler, E., Möhring, R.H., Wünsch, G.: Minimizing total delay in fixed-time controlled traffic networks. In: Fleuren, H., den Hertog, D., Kort, P. (eds.) Operations Research Proceedings 2004, pp. 192–199. Springer, Heidelberg (2005)
15. Liberti, L., Amaldi, E., Maculan, F.M.N.: Mathematical models and a constructive heuristic for finding minimum fundamental cycle bases. Yug. J. of Oper. Res. 15(1), 15–24 (2005)
16. Liebchen, C.: Periodic Timetable Optimization in Public Transport. dissertation.de – Verlag im Internet (2006)
17. Liebchen, C., Wünsch, G.: The Zoo of tree spanner problems. Technical Report 2006-013, TU Berlin, Mathematical Institute (2006)
18. Liebchen, C., Wünsch, G.: Manuscript (2007)
19. Liebchen, C., Wünsch, G., Köhler, E., Reich, A., Rizzi, R.: Benchmarks for strictly fundamental cycle bases. Preprint 003/2007, TU Berlin, Mathematical Institute (2007)
20. Paton, K.: An algorithm for finding a fundamental set of cycles of a graph. Communications of the ACM 12(9), 514–518 (1969)
21. Robertson, N., Seymour, P.D.: Graph minors. V. Excluding a planar graph. J. Comb. Theory, Ser. B. 41(1), 92–114 (1986)
22. Schrijver, A.: Combinatorial Optimization. Algorithms and Combinatorics, vol. 24, Springer, Heidelberg (2003)

A Primal Branch-and-Cut Algorithm for the Degree-Constrained Minimum Spanning Tree Problem

Markus Behle[1], Michael Jünger[2,⋆], and Frauke Liers[2,⋆]

[1] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
[2] Universität zu Köln, Institut für Informatik, Pohligstrasse 1, 50969 Köln, Germany

Abstract. The degree-constrained minimum spanning tree (DCMST) is relevant in the design of networks. It consists of finding a spanning tree whose nodes do not exceed a given maximum degree and whose total edge length is minimum. We design a primal branch-and-cut algorithm that solves instances of the problem to optimality. Primal methods have not been used extensively in the past, and their performance often could not compete with their standard 'dual' counterparts. We show that primal separation procedures yield good bounds for the DCMST problem. On several instances, the primal branch-and-cut program turns out to be competitive with other methods known in the literature. This shows the potential of the primal method.

1 Introduction

In recent years the development of networks in the area of telecommunication and computers has gained much in importance. One of the main goals in the design process is to reach total connectivity at minimum cost. Furthermore, additional constraints must be met, such as a restricted number of connections to a physical unit. Similar problems arise in the planning of road maps, where intersections can only be established among a small number of roads. In the field of integrated circuit design we are faced with the constraint that the number of wired connections to an electronic component is limited. The production of the circuit board shall be at minimum cost.

Problems of this kind can be modelled as the *degree-constrained minimum spanning tree* problem (or *DCMST problem* for short). Network devices, intersections, or electronic components are represented as nodes in a graph. Cables, roads or wires are represented by edges. A cost might be associated with an edge that models, e.g., the cable length or production cost. The technical restriction that the number of connections at a node is bounded is modeled by introducing constraints that bound the node degrees. Garey and Johnson [11] proved that the resulting degree-constrained minimum spanning tree problem is $\mathcal{NP}$-hard.

We are interested in exact solutions for instances of the DCMST problem with a 'primal' branch-and-cut method. Standard branch-and-cut methods can be viewed as 'dual' approaches in which cutting planes are added to iteratively improve the relaxation of the problem. In contrast, in a primal approach cutting planes are added to prove

⋆ Partially supported by the Marie Curie RTN Adonet 504438 funded by the EU and by the German Science Foundation under contracts Ju 204/8 and Li 1675/1 (FL).

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 379–392, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the optimality of the best known feasible solution or to find a better solution. A primal approach has the advantage that the corresponding separation algorithms are often conceptionally easier than their dual versions. Furthermore, at any time during the run of the algorithm a feasible solution of the problem is known. Despite these advantages, primal methods have not yet received much attention in the literature. A possible reason is that the quality of the cutting planes depends on a known feasible solution. Up to now, in most cases the performance of primal approaches could not compete with the performance reachable by standard dual methods.

In this work we revise the primal cutting plane algorithm that Letchford and Lodi introduced in [20,22,21]. To the best of our knowledge, this algorithm has not been implemented before for the solution of practical problems. For the DCMST, our aim is to study the potential of the method by primal separation of known valid inequalities in a branch-and-cut algorithm. Some of the designed primal separation routines are asymptotically faster and much easier to implement than their dual versions. We also present a strategy for branching on a variable in the primal context.

The computational results show that the generated cutting planes are strong. On several classes of instances, the primal method outperforms the genetic algorithms that have been used by other authors. Furthermore, on some classes of instances the performance of the primal approach is comparable to a standard branch-and-cut method used by Raidl [30]. For the standard branch-and-cut algorithm in [6] no computational results on instances from the literature are reported.

In Section 2 we introduce the model. In Section 3 we explain the concept of primal separation together with a primal branch-and-cut algorithm for the DCMST problem. Finally, we show experimental results in Section 5 and discuss directions for further research.

2 Degree-Constrained Minimum Spanning Tree Problem

We are given an undirected, connected graph $G = (V, E)$ with n nodes and m edges. For each edge $e \in E$ a cost $c_e \in \mathbb{R}$ is given. A minimum spanning tree of G is a connected acyclic subgraph $T = (V, E_T)$ that contains all nodes of G and has minimum cost $c(T) := \sum_{e \in E_T} c_e$ among all spanning trees. We define the neighbourhood of a node $i \in V$ as $\delta(i) := \{j \in V \mid \exists\, e = (i, j) \in E\}$. $|\delta(i)|$ is the degree of i in G. With $|\delta_T(i)|$ we denote the degree of i in the spanning tree T. To every node i we associate a capacity $b_i \in \mathbb{Z}$, where $1 \leq b_i \leq n - 1$. The *degree-constrained minimum spanning tree problem* asks for a minimum spanning tree T that satisfies the degree constraint $|\delta_T(i)| \leq b_i$ for all $i \in V$.

In the general setting, finding a DCMST is a hard task, and several, mainly heuristic, solution approaches exist in the literature. Narula and Ho [24] present a heuristic in which first a tree satisfying the degree constraints is generated that is not necessarily minimum. Then a local edge-exchange heuristic is called in order to improve on the cost of the tree while maintaining the node constraints. In a complementary approach, they start by constructing a minimum spanning tree and then repair the violation of degree constraints by exchanging edges. By the use of similar ideas, Savelsbergh and Volgenant [33] get better results than those cited in [24]. Ribeiro and Souza [32]

implement a variable neighbourhood search for generating good heuristic solutions for the DCMST. Recently, Andrade, Lucena and Maculan [1] presented a fast and effective heuristics for its solution. Using Lagrangian dual information, optimality of a solution can be proven in several cases. Goemans [13] designs a polynomial-time approximation algorithm for the DCMST problem with degree bound b for all vertices that finds a spanning tree of maximum degree at most $b + 2$ whose cost is at most the cost of the optimum spanning tree of maximum degree b. Volgenant [36], Knowles and Corne [17], Krishnamoorthy, Ernst and Sharaiha [18] and Raidl [31] present and compare several heuristics, simulated annealing approaches, evolutionary algorithms, Lagrangian relaxation and branch-and-bound methods on different classes of instances. Raidl [30] and Caccetta and Hill [6] also implement a standard branch-and-cut algorithm that solves the DCMST problem exactly. In this work, we are also interested in exact solutions. However, instead of solving the problem by standard branch-and-cut, we exploit the advantages of primal separation in a primal branch-and-cut approach. In the following section, we introduce the notion of primal separation and explain the primal separation algorithms.

3 Primal Separation

Let P_I be the polytope defined as the convex hull of all incidence vectors of feasible solutions of the problem. In a standard cutting plane procedure, we start by optimizing the objective function over a set of equations and inequalities that define a polytope that contains P_I and iteratively improve the relaxation of the problem by adding inequalities ("cutting planes"). To this end, we have to solve the standard separation problem that can be formulated as follows.

Standard separation problem. Given a point $x^* \in \mathbb{R}^m$. Return an inequality valid for P_I that is violated by x^* or prove that none exists.

Grötschel, Lovász and Schrijver [15], Karp and Papadimitriou [16] and Padberg and Rao [27] showed that an optimization problem is polynomial time equivalent to its separation problem. In contrast to an *exact* procedure, a heuristic separation algorithm not necessarily finds a violated inequality if one exists.

In the primal context, we are additionally given $\bar{x} \in \mathbb{R}^m$, a vertex of P_I. Usually, $\bar{x}$ represents the best known primal feasible solution. We formulate the primal separation problem as follows.

Primal separation problem. Given a point $x^* \in \mathbb{R}^m$ and a vertex $\bar{x}$ of P_I. Return an inequality valid for P_I that is violated by x^* and is tight at $\bar{x}$, or prove that none exists.

Padberg and Grötschel [25] showed that the primal separation problem is not harder than its standard version. For 0/1 polytopes, i.e., polytopes in which all vertices are vectors in $\{0, 1\}^m$, also the reverse holds true: Grötschel and Lovász [14] and Schulz, Weismantel and Ziegler [34] proved that the 0/1 optimization problem is polynomial time equivalent to the 0/1 augmentation problem. In the latter we are given a 0/1 point and either want to find a better feasible solution or prove that the given point is optimum. Finally, Eisenbrand, Rinaldi and Ventura [10] showed that the 0/1 augmentation

problem can be reduced to the primal 0/1 separation problem. For combinatorial optimization problems such as the DCMST problem, it follows that standard and primal separation are polynomial time equivalent.

Primal cutting plane algorithms have not yet received a lot of attention. In the early 1960s, Ben-Israel and Charnes used Gomory's method to develop a primal cutting plane algorithm for integer programming that was later simplified by Young [38]. Glover [12] and Arnold and Bellmore [4,3,2] modified the algorithm in order to reduce the number of degenerate pivots. However, only very small toy problems could be solved in practice. Sharma and Sharma [35] tried to improve the method but nevertheless could not compete with dual cutting plane algorithms. The first primal separation routines were developed in 1980 by Padberg and Hong [26] for the travelling salesman problem. In 1990, Barahona and Titan [5] and De Simone and Rinaldi [7] successfully applied primal cutting plane algorithms to the max-cut problem.

With the completion of the proof chain for the equivalence of the 0/1 primal and standard separation [10] and the work of Letchford and Lodi [20,22,21], the interest in primal separation increased again.

3.1 Primal Separation Algorithms for the Degree-Constrained Minimum Spanning Tree Problem

The DCMST problem can be formulated as the following 0/1 integer program.

$$\begin{aligned}
&\min\ c^T x \\
&\text{s.t.} \sum_{e \in E(S)} x_e \le |S| - 1\ \forall\, S \subseteq V,\ |S| \ge 2 && (1)\\
&\sum_{e \in E} x_e = |V| - 1 && (2)\\
&\sum_{e \in \delta(i)} x_e \le b_i \qquad \forall\, i \in V && (3)\\
&x_e \in \{0,1\} \qquad \forall\, e \in E, && (4)
\end{aligned}$$

where $E(S) := \{ij \in E \mid i, j \in S\}$.

Discarding the *node degree constraints* (3) leads to the minimum spanning tree problem which can be solved in polynomial time with e.g., the algorithms of Prim [29] and Kruskal [19]. Thus the node degree constraints (3) are well suited for a Lagrangian relaxation approach that Volgenant applied in [36]. Let the minimum spanning tree (MST) polytope consist of the convex hull of the incidence vectors of spanning trees. Its dimension is $m - 1$. For a complete graph, constraints (1) and $x_e \ge 0$ define facets. The *cycle elimination constraints* (1) are facets iff $|S| = 2$ and the graphs $(S, E(S))$ and $(V \setminus S, E(V \setminus S))$ are connected or $|S| \ge 3$ and the graph $(S, E(S))$ is 2-connected. Edmonds [8,9] showed that the inequalities introduced above are sufficient to describe the MST polytope.

In addition to the cycle elimination constraints (1) we consider two more classes of valid inequalities. The *connectivity constraints*

$$\sum_{e \in \delta(S)} x_e \geq 1 \quad \forall\, S \subset V\colon\; 2 \leq |S| < |V|, \tag{5}$$

where $\delta(S) := \{ij \in E \mid i \in S, j \notin S\}$, assure that the graph is connected. The restriction of the sets S in (1) to cycles $(C, E(C))$ leads to the *specialized cycle elimination constraints*

$$\sum_{e \in E(C)} x_e \leq |C| - 1 \quad \forall\, C \subset V\colon\; |C| < |V| \text{ and } \big(C, E(C)\big) \text{ is a cycle} \tag{6}$$

We start with the relaxation consisting of equation (2), inequalities (3) and the relaxed integrality constraints $0 \leq x_e \leq 1\ \forall\, e \in E$. In the following we present exact primal separation algorithms for the constraint classes (1), (5) and (6) and compare them with their standard versions.

Connectivity Constraints. W.l.o.g. we only consider sets S for which the induced subgraph $(S, E(S))$ is connected. For the standard separation of (5), we temporarily set the edge costs according to x^*. If the value of the minimum cut in the corresponding graph is smaller than 1, a violated inequality is found. The runtime of this separation procedure depends on the chosen min-cut algorithm, e.g. is $O(mn + n^2 \log n)$ for the Nagamochi-Ibaraki [23] algorithm.

For the primal separation, we are additionally given a feasible solution $\bar{x}$ that corresponds to a spanning tree T in G. Candidates are only those sets $S \subset V$ such that $\sum_{e \in \delta(S)} \bar{x}_e = 1$ holds. Every edge e of the spanning tree T induces one such set S. Let $ij = e$ be an edge of T. Temporary deletion of e yields two trees, say T_i with root i and T_j with root j. Let $V_i \subset V$ resp. $V_j \subset V$ be the subset of all nodes that are incident to an edge of T_i resp. T_j. Since T is a spanning tree we have $V_i \,\dot{\cup}\, V_j = V$. W.l.o.g. set $S = V_i$. The primal separation for the connectivity constraints (5) is then the following.

For each of the $n-1$ edges ij of T compute the induced set V_i via depth first search in T. If the value $\sum_{e \in \delta(S_i)} x^*_e$ of the cut $\delta(S_i)$ is less than 1, a violated inequality is found. The runtime of the primal separation of connectivity constraints is $O(n(m+n))$, which is asymptotically faster than its standard counterpart.

Specialized Cycle Elimination Constraints. (6) can be written $\sum_{e \in E(C)} (1 - x_e) \geq 1$. In the standard separation, we temporarily set the edge cost of an edge e to $c'(e) = 1 - x^*_e$. A cycle with cost less than 1 in the corresponding graph determines a violated inequality in the standard sense. Thus, the separation routine amounts to computing for every edge $e = ij$ a shortest path from i to j. If we use Dijkstra's algorithm, the standard separation runs in time $O(mn^2)$.

For the primal separation of (6), we search for cycles C in G that additionally satisfy $\sum_{e \in E(C)} \bar{x}_e = |C| - 1$. Therefore, these cycles contain exactly one edge that is not part of the spanning tree given by $\bar{x}$. This observation immediately gives us the primal separation routine:

For every edge $e = ij$ in $E \setminus T$ find the path R from i to j in T by simultaneously going up level by level in the tree, starting from i and j, until a common predecessor is found. If the cost $c'(e) + c'(R)$ of the fundamental cycle $R \cup e$ is smaller than 1, a violated inequality is found. Primal separation for the specialized cycle elimination constraints runs in $O((m-n)n)$, which is again faster than the standard separation.

Cycle Elimination Constraints. In the case of the traveling salesman problem in which all node degrees are forced to two, the cycle elimination constraints correspond to the subtour elimination constraints for which primal separation is asymptotically faster than its dual version, [26]. In the more general context of degree constrained spanning trees, Padberg and Wolsey presented in [28] a polynomial separation routine for constraints (1). An appropriate network for minimizing the submodular function $g(S) = |S| - \sum_{e \in E(S)} x_e^*$ is defined. In this network $n-2$ min-cuts are computed via max-flow calculations. Using the maximum-distance version of the Goldberg-Tarjan algorithm for the max-flow computations leads to an $O(n^4)$ runtime of the standard separation procedure.

For the primal separation routine, we modify the standard separation. We define a new submodular function $f : S \longrightarrow \mathbb{R}$ for $\emptyset \neq S \subseteq V$ which penalizes sets S that do not satisfy the condition $\sum_{e \in E(S)} \bar{x}_e = |S| - 1$. Let

$$f(S) = \begin{cases} g(S) & \text{if } \sum_{e \in E(S)} \bar{x}_e = |S| - 1 \\ > 1 & \text{otherwise} \end{cases}$$

Primal separation then amounts to finding a set S that minimizes $f(S)$. If the minimum is less than 1, we found a violated inequality in the primal sense.

We define f as

$$f(S) := |S| - \sum_{e \in E(S)} x_e^* + D\Big(|S| - 1 - \sum_{e \in E(S)} \bar{x}_e\Big)$$

with an appropriate constant $D > 0$. First we determine a large enough value for D such that $D > \sum_{e \in E(S)} x_e^*$ holds. $D := \frac{1}{2}(n^2 - n) + 1$ suffices.

It is not hard to see that f is submodular. Following the idea of Padberg and Wolsey we define a network (G^*, c') to find a set S minimizing f. G^* consists of the nodes V, a source node 0 and a sink node $n+1$. For all nodes $i \in V$ let $l_i := \sum_{e \in \delta(i)} (x_e^* + D\bar{x}_e)$. The edges of G^* are constructed as follows.

- For every edge $e = ij$ of E we add the directed edges (i, j) and (j, i) in G^* with costs $c'_{(i,j)} = c'_{(j,i)} = \frac{1}{2}(x_e^* + D\bar{x}_e)$.
- From the source 0 we add for every node $i \in V$ a directed edge $(0, i)$ with cost $c'_{(0,i)} = \max\{\frac{1}{2} l_i - (D+1), 0\}$.
- From every node $i \in V$ we add a directed edge $(i, n+1)$ to the sink $n+1$ with cost $c'_{(i,n+1)} = \max\{(D+1) - \frac{1}{2} l_i, 0\}$.

In the network G^* we consider a $(0, n+1)$-cut given by a set $S \subseteq V$ and calculate its costs $c'\big(S \cup \{0\} :\ (V \setminus S) \cup \{n+1\}\big)$. Let $L := \sum_{i \in V} \max\{\frac{1}{2} l_i - (D+1), 0\}$. The value of a $(0, n+1)$-cut in G^* is

$$c'\big(S \cup \{0\} :\ (V \setminus S) \cup \{n+1\}\big)$$

$$= \sum_{i \in V \setminus S} \max\{\tfrac{1}{2} l_i - (D+1), 0\} + \sum_{i \in S} \max\{(D+1) - \tfrac{1}{2} l_i, 0\} + c'(S :\ V \setminus S)$$

$$= (D+1)|S| - \tfrac{1}{2}\sum_{i\in S} l_i + \tfrac{1}{2} \sum_{\substack{e=ij \\ i\in S, j\in V\setminus S}} (x_e^* + D\bar{x}_e) + L$$
$$= f(S) + D + L$$

Thus, determining a set S which minimizes f amounts to minimizing the $(0, n+1)$-cut in (G^*, c'). Since S may not be the empty set, following Padberg and Wolsey, we need $n-2$ max-flow calculations to compute a minimum $(0, n+1)$-cut. The runtime of the primal separation asymptotically equals that of the standard separation. Padberg and Wolsey already noted the fact that the algorithm for eliminating the cycles is quite expensive, given that determining minimum spanning trees is an easy task. Furthermore, the primal separation routine for the cycles is neither asymptotically faster nor conceptionally easier than its standard version.

4 Primal Branch-and-Cut Algorithm

In this section, we describe a primal branch-and-cut algorithm for the DCMST problem. We use a modified version of the algorithm introduced by Letchford and Lodi in [21]. The algorithm starts with a feasible solution $\bar{x}$ generated with Narula and Ho's primal heuristic [24] which is a modified version of Prim's algorithm for MST. Subsequently, we improve the solution by the variable neighbourhood search [32].

For DCMST, the initial relaxation consists of the spanning tree equation (2), the node degree constraints (3) and the relaxed integrality constraints, see Section 3.1. Generated violated inequalities are added to the current relaxation of the problem. In case the separation procedures fail to determine a violated inequality, the algorithm branches. The following section introduces a primal branching rule.

4.1 Primal Branching Rule

In a primal branch-and-cut algorithm, we need in each subproblem a solution $\bar{x}$ that is feasible for it. Applying the standard rules for branching on a binary variable usually destroys feasibility of $\bar{x}$ in one of the two branches. Letchford and Lodi [21] presented a primal branching rule maintaining feasibility for the special case that $\bar{x}$ is identically **0** and all constraints in the relaxation are tight at $\bar{x}$. The former is no restriction for 0/1 integer linear programs. Relaxing the above mentioned restrictions, the branching rule in the spirit of Letchford and Lodi reads as follows. Let N_0 and N_1 be the set of indices of variables that are set to 0 resp. 1. Let a relaxation P be given as $P = \{x : Ax \leq b, 0 \leq x \leq 1\}$, and $\bar{x} \in P \cap \{0,1\}^n$. Branching according to N_0 and N_1 leads to P^1 which is the intersection of P with the subspaces of the variables x_j fixed to 0 resp. 1 for $j \in N_0, N_1$

$$P^1 = \left\{ x : \begin{array}{cr} Ax \leq b & \\ 0 \leq x_j \leq 1 & \forall j \notin N_0 \cup N_1 \\ x_j = 0 & \forall j \in N_0 \\ x_j = 1 & \forall j \in N_1 \end{array} \right\}$$

We maintain feasibility of $\bar{x}$ by optimizing in each subproblem over the convex hull of $P^1 \cup P^2$, where $P^2 := \{x : x = \bar{x}\}$. Let $\bar{a}_k^1 := \sum_{j:\bar{x}_j=1} a_{kj}$. In the primal branching rule, we distinguish three cases.

1. (a) $\exists i : \bar{x}_i = 0 \wedge i \in N_1$. Set $y := x_i$.
   (b) $\exists i : \bar{x}_i = 1 \wedge i \in N_0$. Set $y := 1 - x_i$.
   Then

$$\operatorname{conv}(P^1 \cup P^2) = \left\{ x : \begin{array}{lr} \sum_j a_{kj}x_j \leq \bar{a}_k^1 - (\bar{a}_k^1 - b_k)y & \forall k \\ 0 \leq x_j \leq y & \forall j : \bar{x}_j = 0 \wedge j \notin N_0 \cup N_1 \\ x_j = 0 & \forall j : \bar{x}_j = 0 \wedge j \in N_0 \\ x_j = y & \forall j \neq i : \bar{x}_j = 0 \wedge j \in N_1 \\ 1 - y \leq x_j \leq 1 & \forall j : \bar{x}_j = 1 \wedge j \notin N_0 \cup N_1 \\ x_j = 1 - y & \forall j \neq i : \bar{x}_j = 1 \wedge j \in N_0 \\ x_j = 1 & \forall j : \bar{x}_j = 1 \wedge j \in N_1 \\ 0 \leq y \leq 1 & \end{array} \right\}$$

2. $\nexists i : \bar{x}_i = 0 \wedge i \in N_1$ and $\nexists i : \bar{x}_i = 1 \wedge i \in N_0$. Then $\operatorname{conv}(P^1 \cup P^2) = P^1$.

Whenever a new branch-and-bound node is created, the subproblem relaxation is modified according to the corresponding case above.

4.2 Outline of the Primal Branch-and-Cut Algorithm

In every node of the branch-and-bound tree valid violated cutting planes are generated. The used cutting plane procedure is a modification of the one developed by Letchford and Lodi in [20]. We describe it in the following.

Let the result of a primal simplex iteration be x^*. In case x^* is feasible for the DCMST problem and improves upon the formerly best known solution, a new degree-constrained spanning tree $\bar{x}$ is found. If $\bar{x}$ is dual feasible, the current subproblem is optimized and we can fathom the node. In case x^* is not binary or contains a cycle, the primal separation routines are called. In contrast to a standard cutting plane algorithm it is possible that no primal separating hyperplane for a binary but infeasible point x^* can be generated. This may happen e.g. if we pivot from $\bar{x}$ to a 0/1 vertex of the relaxation which is feasible for all inequalities that can be generated by primal separation. In this case we aim at cutting off the infeasible point by standard separation routines. In case of failure, we separate a fractional point x^* by a Chvátal-Gomory cut that is tight at $\bar{x}$ and only contains integral coefficients. So at all times our matrix A is integral.

We generate Chvátal-Gomory cuts in the following way. Without loss of generality, we consider a linear relaxation of the form $P = \{x : Ax \leq b, x \geq 0\}$ with $A \in \mathbb{Q}^{m \times n}$. Let $A_{\cdot k}$ denote the k-th column of A. For $u \in \mathbb{Q}^m_{\geq 0}$ the inequality $\sum_{k=1}^n \lfloor u^T A_{\cdot k} \rfloor x_k \leq \lfloor u^T b \rfloor$ is valid for P_I, see e.g. [37]. The coefficients of this Chvátal-Gomory cut are integer. For each row i of the inverse of the basis matrix A_B we define u_i as the fractional part of that row, i.e. $u_i := e_i^T A_B^{-1} - \lfloor e_i^T A_B^{-1} \rfloor$. Now for each u_i we generate the Chvátal-Gomory cut and check if it is tight at $\bar{x}$.

The primal cutting plane procedure for the solution of a node in the branch-and-bound tree for the DCMST problem works as follows. Its generalization to other problems is obvious.

- Step 1: Compute a primal feasible basis for the basic solution $\bar{x}$.
- Step 2: Perform a primal simplex pivot. Let x^* be the new solution.
- Step 3: If x^* is binary and represents a DCMST, set $\bar{x} = x^*$. If $\bar{x}$ is optimal, fathom the node. Otherwise go to Step 2.
- Step 4: Call the primal separation. If violated inequalities are generated, pivot back to $\bar{x}$, add them to the LP and go to Step 2.
- Step 5: If x^* is binary but contains a cycle, generate a separating hyperplane that is not tight at $\bar{x}$, add it to the LP and go to Step 1.
- Step 6: Determine a Chvátal-Gomory cut that is tight at $\bar{x}$ and only contains integral coefficients. If such a cut is found, pivot back to $\bar{x}$, add it to the LP and go to Step 2. Otherwise branch according to the primal branching rule outlined in section 4.1.

The difference between this algorithm and the one introduced by Letchford and Lodi mainly lies in Step 5. It is called Step 5b in their enhanced algorithm. In case a binary but infeasible point x^* is found, Letchford and Lodi generate a cutting plane that is not tight at $\bar{x}$. Then the dual simplex algorithm is used to compute a new fractional point $\hat{x}$ that is a convex combination of $\bar{x}$ and x^*. Then they set $x^* := \hat{x}$ and remove the just inserted cutting plane as the mixed-integer cut generated in the next step will cut off $\hat{x}$ and thus also the old binary solution.

In contrast, for esthetical reasons we decided not to switch to the dual simplex but to always stick to its primal version throughout the computations. We also cut off the infeasible point x^* with a standard cutting plane and thus lose primal feasibility of the tableau. As we need to stay at $\bar{x}$, we compute in Step 1 an appropriate primal feasible basis. To this end, we temporarily introduce a new objective function c' with entries $c'_i := 1 - 2\bar{x}_i$. Minimizing the new objective function by the primal simplex algorithm yields the optimum solution $\bar{x}$ and a primal feasible tableau. Replacing the objective function by c again keeps its primal feasibility.

5 Experimental Results

We implemented the primal branch-and-bound algorithm from Section 4.2 and the primal separation routines outlined in Section 3.1 in a straightforward way. When primal separation is called in Step 4, we first try to find a violated specialized cycle elimination constraint (6). If we do not find one we try to find a violated connectivity constraint (5) and if we still do not succeed, we try to separate a cycle elimination constraint (1). For performance reasons we do not separate cycle elimination constraints in the way we describe in 3.1 but set up a pool of cycle elimination constraints tight at $\bar{x}$ whenever we find a new $\bar{x}$. We do a pool separation. The size of this pool depends on the problem structure and varies from 10000 up to 200000 cuts.

In Step 5 we use the standard separation for the specialized cycle elimination constraints (6). If we arrive at Step 5, the binary solution contains a cycle that the primal

separation was not able to separate. So we will always find a violated cut in Step 5. In case that none of the former separation routines found a cut we invoke the Chvátal-Gomory separation. If still no cut can be found or if the number of Chvátal-Gomory cuts in the actual node of the branch-and-bound tree exceeds 10, we branch. This prevents primal degeneracy.

We took instances from Knowles and Corne [17] and Krishnamoorthy, Ernst and Sharaiha [18] which are grouped into the 6 classes R, M, CRD, SYM, STR and SHRD. From these classes, we took all instances with at most 100 nodes resulting in 234 instances in total. All computations were done on a Pentium 4 processor under Linux. We use CPLEX 9.0 as linear program solver. The quality of the solution is measured by the gap, i.e. the difference of the primal and the dual bounds divided by the primal one. For each instance, we set an upper limit of four hours of cpu time. Within this time interval, 228 out of the 234 instances could be solved to optimality. For the remaining 6 instances, the gaps are always smaller than 6.3%. In the tables n is the number of

Table 1. R and M instances

		n	b	Type	PBC							SBC	GA
					B&B nodes	height	Pcuts	Scuts	CGcuts	time	Q	Q	Q
R		50	5	n1	1	0	8	0	0	1.62	1.61	-	1.74
R		50	5	n2	1	0	26	6	1	1.61	1.69	-	1.83
R		50	5	n3	1	0	11	3	1	3.59	1.62	-	1.78
R		100	5	n1	21	5	45	178	10	94.16	1.61	-	1.73
R		100	5	n2	1543	24	80	4144	182	5525.55	1.50	-	1.60
R		100	5	n3	2	1	22	16	1	14.08	1.55	-	1.69
M	*	50	5	n1	172	34	1690	105	141	14400	2.45	2.45	3.15
M		50	5	n2	3	1	707	17	0	591.82	2.21	2.21	2.99
M		50	5	n3	12	4	615	168	22	130.07	2.36	2.36	2.58
M		100	5	n1	14	7	251	223	11	1471.3	1.98	1.98	2.44
M		100	5	n2	1580	32	1299	10705	676	10950.1	2.08	2.08	2.84
M		100	5	n3	160	29	710	1246	111	2041.48	1.98	1.98	2.86

nodes. All nodes have the same upper bound b on the node degree. Our algorithm can also handle individual node degree constraints. The time spent in the branch-and-cut process is given in seconds. The time spent in the starting heuristics that generate a good feasible solution $\bar{x}$ is usually less than a second. The primal branch-and-cut algorithm is named *PBC*. *B&B nodes* shows the number of branch-and-bound nodes and *height* the height of the branch-and-bound tree. The number of generated violated primal, standard resp. Chvátal-Gomory cuts are given in the columns *Pcuts*, *Scuts* resp. *CGcuts*.

The R and M instances are taken from Knowles and Corne [17]. Their *GA* heuristic performs well on these instances. Results are averaged over 20 passes and are given as a quotient Q which is the cost of the found degree-constrained spanning tree divided by the cost of the minimum spanning tree. Running times are not published. The primal PBC algorithm solves the R instances (see table 1) with 50 nodes in less than 4 seconds to optimality. For the larger instances the program has to branch. The class of the M

instances is constructed by Knowles and Corne such that the minimum spanning tree and the DCMST are very different which causes easy greedy heuristics to fail. The M instances (see table 1) were also tackled by Raidl [30] with a standard branch-and-cut approach which we name *SBC*. However, a direct comparison between the PBC and SBC algorithms is difficult as Lagrangian relaxation was used in the latter. Using the PBC algorithm, 5 out of the 6 instances could be solved to optimality within the given time interval. For the instance marked with an asterisk the gap at the time limit was 4.1%. With their heuristics, Andrade, Lucena and Maculan [1] can solve the same R and M instances in less than 30 seconds each and furthermore prove optimality of their solution.

Table 2. CRD, SYM and STR instances

			PBC						BB
type	n	b	B&B nodes	height	Pcuts	Scuts	CGcuts	time	Gap
CRD	30	3	1	0	4.5	0	0	0.11	0.70
	50	3	1	0	10.8	0	0	0.42	0.01
	70	3	3.4	1.6	42.5	13.5	24.5	54.54	0.01
	100	3	5.89	1.11	115.4	5.9	48.8	290.8	0.04
SYM	30	3	1	0	5	0	1.7	0.49	4.26
	30	4	1	0	4	0	0.7	0.17	0.16
	50	3	33.3	4.8	41.9	24.4	317	124.94	4.78
	50	4	142.4	3.6	34.5	15.7	147.8	107.91	0.95
	50	5	1	0	1.6	0	0	0.11	0.10
	70	3	36	5.2	37.8	9.4	361.8	77.35	5.60
	70	4	174.7	4.3	31.3	20.9	522.1	284.93	1.03
	70	5	153.2	2.5	15.3	12.2	160.6	219.37	0.11
STR	30	4	1	0	25.2	0	0	0.2	13.45
	30	5	1	0	28.8	0	0	0.28	10.8
	50	4	2.6	0.4	96.6	2.2	18.9	4.3	12.31
	50	5	3.4	0.6	105.2	2.5	26.4	4.8	9.88
	70	4	1.7	0.4	268	0.3	11.5	24.17	11.49
	70	5	9.2	0.4	263.3	2.9	88.2	38.75	9.21
	100	3	5.7	0.8	274.3	0.2	3.3	1251.1	12.85
	100	4	1	0	544.63	0	0	171.84	10.59
	100	5	1	0	555.25	0.13	0.75	79.45	8.48

The four instance classes CRD, SYM, STR and SHRD are taken from Krishnamoorthy, Ernst and Sharaiha [18]. We compare our algorithm with their branch-and-bound approach called *BB* with a time limit of 600 seconds. The values in the table 2 are the means of 10 different instances with the same number of nodes and the same b. The CRD instances are two-dimensional Euclidean graphs. The SYM instances are random multi-dimensional Euclidean graphs. Usually these instances are solved within seconds with PBC. One CRD instance with 100 nodes exceeded the time limit reaching a gap of $3.4 \cdot 10^{-3}\%$. Instances that exceeded the time limit were excluded from the averages. The instances of the STR class usually cannot be solved to optimality

by the branch-and-bound approach of Krishnamoorthy, Ernst and Sharaiha with a time limit of 600 seconds [18]. Except for 4 large instances out of the 100, we can solve all instances to optimality. The gaps for these 4 instances range from 3.2% to 6.3%. 90 instances could be solved within 600 seconds. The small instances are solved within seconds.

Krishnamoorthy, Ernst and Sharaiha claim that the instances from the SHRD class are the hardest since none of its members could be solved with their branch-and-bound approach within 600 seconds. We compare our results with their best heuristic *PSS*. In contrast to PSS, Raidl's approach [30] and our algorithm did not have any problems (see table 3). We could solve every instance to optimality in less than 21 seconds. Again, with their heuristics, Andrade, Lucena and Maculan are able to solve SHRD instances with up to 309 nodes in less than 30 seconds per instance. They can additionally prove optimality.

Table 3. SHRD instances

		PBC						PSS
n	b	B&B nodes	height	Pcuts	Scuts	CGcuts	time	Gap
15	3	1	0	3	4	0	0.06	1.72
15	4	1	0	0	0	0	0.26	2.08
15	5	1	0	0	0	0	0.18	0
20	3	2	1	18	16	0	1.53	0.45
20	4	1	0	5	1	1	2	0
20	5	1	0	0	0	0	1.58	0.47
25	3	1	0	1	0	0	5.73	0
25	4	1	0	2	1	0	4.82	0
25	5	1	0	0	0	0	1.87	1.07
30	3	4	2	302	33	2	20.25	0
30	4	2	1	15	16	1	7.55	0
30	5	2	1	47	16	0	4.02	0

As can be expected, the quality of the starting feasible solution given by a heuristic is important for the performance of the primal branch-and-cut process. Usually, the closer the objective value of the solution is to the optimal objective value, the faster the process terminates. The usage of the primal cutting planes varies from instance to instance, not only from class to class. For some instances it is useful to spend more time in the separation whereas others are solved faster if less cutting planes are generated and more branching is done. The special cycle elimination constraints and the connectivity constraints can be separated very fast but are usually weaker than the cycle elimination constraints that turned out to be the most important constraints for the tested instances. We do not perform an exact separation of this class but set up a pool of constraints that is searched for a violated one. We observed that the larger the pool is, the less standard and Chvátal-Gomory cuts have to be generated. As we do not refill this pool until a new $\bar{x}$ is found, the longer we stay at a certain $\bar{x}$ the more standard and Chvátal-Gomory cuts are needed. This effect can be observed for the large R and M instances.

6 Conclusion

In this paper we presented primal separation routines for the DCMST problem and implemented them in primal branch-and-cut procedure. We showed that primal separation can be conceptionally easier and theoretically faster than the standard dual separation. The computational results show that primal methods can compete with existing approaches. Research has been undertaken in more depth for dual than for primal methods yet, and several questions remain to be answered in the primal context such as, e.g., what are the most effective branching rules, rules for chosing the branching variables, etc. We believe that there is potential for primal branch-and-cut methods to become competitive with dual methods also for other hard problems.

Acknowledgments

The authors thank A. Letchford and A. Lodi for fruitful discussions.

References

1. Andrade, R., Lucena, A., Maculan, N.: Using lagrangian dual information to generate degree constrained spanning trees. Discrete Applied Mathematics 154(5), 703–717 (2006)
2. Arnold, L.R., Bellmore, M.: A bounding minimization problem for primal integer programming. Operations Research 22, 383–392 (1974)
3. Arnold, L.R., Bellmore, M.: A generated cut for primal integer programming. Operations Research 22, 137–143 (1974)
4. Arnold, L.R., Bellmore, M.: Iteration skipping in primal integer programming. Operations Research 22, 129–136 (1974)
5. Barahona, F., Titan, H.: Max mean cuts and max cuts. In: Combinatorial Optimization in Science and Technology, pp. 30–45 (1991)
6. Caccetta, L., Hill, S.P.: A branch and cut method for the degree-constrained minimum spanning tree problem. Networks 37(2), 74–83 (2001)
7. De Simone, C., Rinaldi, G.: A cutting plane algorithm for the max-cut problem. Optimization Methods and Software 3, 195–214 (1994)
8. Edmonds, J.: Submodular functions, matroids, and certain polyhedra. In: Combinatorial Structures and their Applications, pp. 69–87. Gordon and Breach, New York (1970)
9. Edmonds, J.: Matroids and the greedy algorithm. Math. Programming 1, 127–136 (1971)
10. Eisenbrand, F., Rinaldi, G., Ventura, P.: 0/1 optimization and 0/1 primal separation are equivalent. In: Proceedings of the 13th annual ACM-SIAM symposium on discrete algorithms, SODA '02, pp. 920–926 (2002)
11. Garey, M.R., Johnson, D.S.: Computers and Intractability, A Guide to the Theory of NP-Completeness. Freeman, San Francisco (1979)
12. Glover, F.: A new foundation for a simplified primal integer programming algorithm. Operations Research 16, 727–740 (1968)
13. Goemans, M.X.: Minimum bounded-degree spanning trees. In: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 273–282 (2006)
14. Grötschel, M., Lovász, L.: Handbook of Combinatorics, vol. 2, chapter Combinatorial Optimization, pp. 1541–1597. North Holland (1995)
15. Grötschel, M., Lovász, L., Schrijver, A.: The ellipsoid method and its consequences in combinatorial optimization. Combinatorica 1(2), 169–197 (1981)

16. Karp, R.M., Papadimitriou, C.H.: On linear characterizations of combinatorial optimization problems. In: 21st Annual Symposium on Foundations of Computer Science, pp. 1–9, Syracuse, New York (1980)
17. Knowles, J.D., Corne, D.W.: A new evolutionary approach to the degree-constrained minimum spanning tree problem. IEEE Transactions on Evolutionary Computation 4(2), 125–134 (2000)
18. Krishnamoorthy, M., Ernst, A.T., Sharaiha, Y.M.: Comparison of algorithms for the degree constrained minimum spanning tree. Journal of Heuristics 7, 587–611 (2001)
19. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. In: Proceedings of the American Mathematics Society, vol. 7(1) pp. 48–50 (1956)
20. Letchford, A.N., Lodi, A.: Primal cutting plane algorithms revisited. Mathematical Methods of Operations Research 56(1), 67–81 (2002)
21. Letchford, A.N., Lodi, A.: An augment-and-branch-and-cut framework for mixed 0-1 programming. In: Jünger, M., Reinelt, G., Rinaldi, G. (eds.) Combinatorial Optimization - Eureka, You Shrink! LNCS, vol. 2570, Springer, Heidelberg (2003)
22. Letchford, A.N., Lodi, A.: Primal separation algorithms. 4OR 1(3), 209–224 (2003)
23. Nagamochi, H., Ibaraki, T.: Computing edge connectivity in multigraphs and capacitated graphs. SIAM Journal on Discrete Mathematics 5, 54–66 (1992)
24. Narula, S.C., Ho, C.A.: Degree-constrained minimum spanning tree. Computers & Operations Research 7, 239–249 (1980)
25. Padberg, M.W., Grötschel, M.: The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization (chapter). In: Polyhedral computations, pp. 307–360. Wiley, Chichester (1985)
26. Padberg, M.W., Hong, S.: On the symmetric travelling salesman problem: a computational study. Mathematical Programming Study 12, 78–107 (1980)
27. Padberg, M.W., Rao, M.R.: The russian method for linear programming III: Bounded integer programming. Technical Report 81-39, Graduate School of Business and Administration, New York University (1981)
28. Padberg, M.W., Wolsey, L.A.: Trees and cuts. Annals of Discrete Mathematics 17, 511–517 (1983)
29. Prim, R.: Shortest connection networks and some generalizations. Bell System Technical Journal 36, 1389–1401 (1957)
30. Raidl, G.R.: personal communication
31. Raidl, G.R.: An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem. In: Proceedings of the 2000 IEEE Congress on Evolutionary Computation, vol. 1, pp. 104–111 (2000)
32. Ribeiro, C.C., Souza, M.C.: Variable neighborhood search for the degree-constrained minimum spanning tree problem. Discrete Applied Mathematics 118(1-2), 43–54 (2002)
33. Savelsbergh, M., Volgenant, T.: Edge exchanges in the degree-constrained minimum spanning tree problem. Computers & Operations Research 12, 341–348 (1985)
34. Schulz, A.S., Weismantel, R., Ziegler, G.M.: 0/1 integer programming: Optimization and augmentation are equivalent. In: Spirakis, P.G. (ed.) ESA 1995. LNCS, vol. 979, pp. 473–483. Springer, Heidelberg (1995)
35. Sharma, S., Sharma, B.: New technique for solving primal all-integer linear programming. Opsearch 34, 62–68 (1997)
36. Volgenant, A.: A lagrangean approach to the degree-constrained minimum spanning tree problem. European Journal of Operational Research 39, 325–331 (1989)
37. Wolsey, L.A.: Integer Programming. Wiley-Interscience, New York, NY, USA (1998)
38. Young, R.D.: A simplified primal (all-integer) integer programming algorithm. Operations Research 16, 750–782 (1968)

Experimental Analysis of Algorithms for Updating Minimum Spanning Trees on Graphs Subject to Changes on Edge Weights

Celso C. Ribeiro* and Rodrigo F. Toso**

Institute of Computing, Universidade Federal Fluminense
Rua Passo da Pátria 156, Niterói, RJ 24210-240, Brazil
celso@inf.puc-rio.br, rtoso@rutcor.rutgers.edu

Abstract. We consider the problem of maintaining a minimum spanning tree of a dynamically changing graph, subject to changes on edge weights. We propose an on-line fully-dynamic algorithm that runs in time $O(|E|)$ when the easy-to-implement DRD-trees data structure for dynamic trees is used. Numerical experiments illustrate the efficiency of the approach.

Keywords: Minimum spanning trees, dynamic graph algorithms, experimental analysis, algorithms, data structures, DRD-trees.

1 Introduction

The dynamic minimum spanning tree problem is that of maintaining a minimum spanning tree (MST) of a dynamically changing graph $G = (V, E)$, where changes can be vertex insertions and deletions, edge insertions and deletions, or edge weight modifications. The problem is said to be *fully dynamic* if insertion and deletion operations are allowed (or if the edge weights can increase and decrease). The problem is said to be *partially dynamic* if only one kind of operation is allowed (either deletions or insertions, either weight increases or weight decreases). The problem is said to be *on-line* if the dynamic changes must be processed in real time (i.e., there is no preprocessing and the updates are performed one at a time) [1].

There are several variants of the dynamic minimum spanning tree problem. Spira and Pan [16] proposed algorithms for vertex insertion and vertex deletion variants. Zaroliagis [18] surveyed experimental studies on dynamic graph problems subject to edge insertions and deletions. See also [7,8,9,10,11] for algorithms and [1,5] for experimental studies involving edge insertions and edge deletions.

In this work, we make a step toward the experimental evaluation of algorithms to update a minimum spanning tree after edge weight changes. Such algorithms are needful in the implementation of local search heuristics for solving broadcast

* Supported by CNPq and FAPERJ research grants.
** Supported by a CNPq scholarship.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 393–405, 2007.
© Springer-Verlag Berlin Heidelberg 2007

and design problems in communication networks, similarly to the algorithms for dynamic shortest path problems studied by Buriol et al. [2,3,4] in the context of the weight setting problem in OSPF/IS-IS routing.

In the next section, we describe and evaluate a new and easy-to-implement supporting data structure for dynamic trees representation, called DRD-trees. A new fully-dynamic algorithm for updating a minimum spanning tree after edge weight changes is proposed in Section 3. Complexity issues are also considered in this section. An experimental evaluation of several algorithms is carried out in Section 4 on a comprehensive set of test instances, showing that our approach outperforms the fastest algorithms for real-size instances. Concluding remarks are drawn in the last section.

2 Data Structures: Doubly-Linked Reversed Dynamic Trees

We are given a dynamic graph $G = (V, E)$ with vertex set V, edge set E, and non-negative weights $w(i, j)$ associated with each edge $(i, j) \in E$. Let $T = (V, E')$ be a dynamic minimum weight spanning tree of G, i.e., a minimum spanning tree subject to structural changes caused by modifications in the edge weights.

The dynamic trees problem [15] consists in maintaining a forest of disjoint trees that change over time through edge insertions and deletions. For example, one may want to link two trees by adding an edge or to cut a tree by removing an edge. We denote by $\texttt{Link}(i, j, w)$ (resp. $\texttt{Cut}(i, j)$) the operation of inserting (resp. deleting) edge (i, j) into (resp. from) tree $T = (V, E')$. A simple way to represent a forest of disjoint trees is through a set of rooted, directed trees. To manipulate these trees, the following operations are also made available:

– $\texttt{Root}(i)$: returns the root of the tree containing vertex i;
– $\texttt{Evert}(i)$: makes vertex i the root of its tree; and
– $\texttt{Find_Max}(i)$: returns the max-weight edge in the path from i to $\texttt{Root}(i)$.

Operations $\texttt{Link}(i, j, w)$ and $\texttt{Cut}(i, j)$ can be implemented in constant time using *rooted reversed dynamic trees* (RD-trees): each vertex $i \in V$ stores its parent and the weight of the edge between them. Operations $\texttt{Evert}(i)$, $\texttt{Root}(i)$ and $\texttt{Find_Max}(i)$ run in linear time, since they depend on the length of the path from vertex i to $\texttt{Root}(i)$. However, a link operation may require the execution of an evert operation in the case of undirected trees, therefore resulting in linear running time. Sleator and Tarjan [15] (resp. Henzinger and King [10] and Werneck and Tarjan [17]) proposed the ST-trees (resp. ET-trees and self-adjusting top trees), designed to support all the above operations in logarithmic time (in fact, ET-trees do not support $\texttt{Find_Max}(i)$).

We propose an extension of the RD-trees by building doubly-linked trees (DRD-trees), instead of simply reversed trees. This can be accomplished by an additional list associated with each vertex $v \in V$, storing each of its children.

The motivation for this extension comes from the need to detect if an arbitrary edge $(x, y) \in E$ reconnects the two disjoint subtrees T_i and T_j resulting from

the removal of an edge $(i, j) \in E'$, where T_i (resp. T_j) is the resulting subtree containing vertex i (resp. j). This *connectivity query* can be answered by checking if x and y belong to different subtrees (i.e., if the roots of their subtrees are different). However, storing all vertices adjacent to j and assuming (without loss of generality) that vertex i is the parent of j, one may apply a depth-first search starting from j and label all reachable vertices (i.e., those in T_j) with one, zero otherwise. These labels can then be used to answer the above connectivity query in amortized constant time when the number of queries is $O(n)$, in contrast with the root-based data structures, which depend on the implementation of `Root(`i`)` and take at least logarithmic time.

3 Fully-Dynamic Algorithm

We present a new fully dynamic algorithm to update the minimum spanning tree of a graph subject to edge weight changes. It makes use of DRD-trees, which combine easy implementation with efficiency. Two cases are considered separately: edge weight decreases are considered in Section 3.1, while edge weight increases are handled in Section 3.2.

Externally to both algorithms, we maintain an edge list A sorted from left to right by the non-decreasing order of edge weights. Whenever an edge weight is increased or decreased, this list is updated to reflect the new ordering. Let $A(k)$ be the k-th edge in the ordered list A, with $k = 1, \ldots, |E|$, and $A(i, j)$ be the position of edge $(i, j) \in E$ in the ordered list A. List A is stored as a skip list [14] for improved efficiency. The general framework used to update the minimum spanning tree, maintaining list A correctly ordered, is shown in Algorithm 1.

Algorithm 1. Updates a minimum spanning tree subject to edge weight changes

Input: Graph $G = (V, E)$, weights w.
1: Build a list A with all $(i, j) \in E$;
2: Sort list A by non-decreasing order of weights;
3: Use list A to compute the MST $T = (V, E')$;
4: **while** there is an update to be processed **do**
5: Let $(i, j) \in E$ be the edge whose weight will change to w^{new};
6: $s \leftarrow A(i, j)$;
7: Save the old edge weight: $w^{old} \leftarrow w(i, j)$;
8: Set the new edge weight: $w(i, j) \leftarrow w^{new}$;
9: Reorder list A by non-decreasing edge weights;
10: $f \leftarrow A(i, j)$;
11: **if** $w^{new} < w^{old}$ **and** $(i, j) \notin E'$ **then**
12: Apply Algorithm 2. with parameters T, (i, j), and w;
13: **else if** $w^{new} > w^{old}$ **and** $(i, j) \in E'$ **then**
14: Apply Algorithm 3. with parameters T, (i, j), s, f, and w;
15: **end if**
16: **end while**

The ordered list A is initialized in lines 1 and 2. A minimal spanning tree $T = (V, E')$ is computed in line 3 using Kruskal's algorithm [12]. The loop in lines 4 to 16 is performed until all updates have been processed. The edge to be updated and its new weight are read in line 5. The position s of edge (i, j) in the current list A is saved in line 6, before the list is reordered. The old weight w^{old} of edge (i, j) is saved in line 7, while the new weight w^{new} is set in line 8. The list A is reordered in line 9 after the change of the weight $w(i, j)$ of edge (i, j). To ensure the correctness of the decremental updates, in case the new weight w^{new} of (i, j) is equal to the weight of other edges, then edge (i, j) should be placed in the last position among all edges with the same weight. The position f of edge (i, j) in the reordered list A is saved in line 10. If the comparison in line 11 (resp. in line 13) determines that the new weight of edge (i, j) is smaller (resp. larger) than the old one and (i, j) is a non-tree (resp. tree) edge, then Algorithm 2. (resp. Algorithm 3.) is applied in line 12 (resp. line 14) to update the current minimum spanning tree, since decreasing the weight of a tree edge (resp. increasing the weight of a non-tree edge) does not change the latter. We notice that updates in the weight function are reflected both in list A and in the data structure used to store the minimum spanning tree T.

3.1 Weight Decreases

Whenever the weight of a non-tree edge (i, j) is decreased, one has to find the maximum weight edge (x, y) along the path from i to j in T and to remove it if $w(x, y) > w(i, j)$ [16]. This can be accomplished by using any dynamic tree data structure to store the MST.

The procedure to update the minimum spanning tree is described in Algorithm 2. Vertex i is made the new root of the tree in lines 1 to 3. The maximum weight edge (x, y) in the path from j to i is computed in line 4. If the weight of edge (x, y) is larger than that of edge (i, j) (comparison in line 5), then the former is removed from the tree by the `Cut`(x, y) operation in line 6 and the new edge (i, j) is inserted by the `Link`$(i, j, w(i, j))$ operation in line 7. The updated MST is returned in line 9.

The efficiency of the above computations depend on the underlying structure used to maintain the MST and to implement the path operations. If RD-trees or DRD-trees are used, then Algorithm 2. runs in time $O(|V|)$. It runs in time $O(\log |V|)$ if a more complex implementation (such as ST-trees) is used.

3.2 Weight Increases

We now face the hardest part of the problem. If the weight of a tree edge (i, j) is increased, the latter may have to be removed from the current minimum spanning tree. In this case, one has to find the minimum weight edge connecting the two resulting disjoint subtrees (namely, $T_i = (V_i, E'_i)$ and $T_j = (V_j, E'_j)$) to be inserted in the new minimum spanning tree. There are up to $O(n^2)$ such candidate edges [16].

The procedure to update the minimum spanning tree is described in Algorithm 3. Vertex j is assumed to be a child of vertex i and a depth-first search

Algorithm 2. Non-tree edge weight decreases

Input: MST $T = (V, E')$, non-tree edge (i, j) subject to a weight decrease, weights w.

1: **if** $i \neq$ `Root`(i) **then**
2:    `Evert`(i);
3: **end if**
4: $(x, y) \leftarrow$ `Find_Max`(j);
5: **if** $w(x, y) > w(i, j)$ **then**
6:    `Cut`(x, y);
7:    `Link`$(i, j, w(i, j))$;
8: **end if**
9: **return** updated minimum spanning tree T;

is applied to the current MST from vertex j in line 1. Vertices in T_j are those reachable from j by a depth-first search in T. The position k of the first candidate edge to replace (i, j) is set in line 2. The loop in lines 3 to 11 scans all edges between positions s and f of list A. The next edge (x, y) to be investigated is set in line 4 as that in position k of list A. If vertices x and y are in different subtrees (comparison in line 5), then edge (i, j) is eliminated from the current tree in line 6 and the two subtrees T_i and T_j are linked by edge (x, y) in line 7. The updated minimum spanning tree is returned in line 8. Otherwise, the current position in list A is incremented by one in line 10 and a new edge is examined. If no improving edge can be found to replace edge (i, j), then the unchanged current minimum spanning tree is returned in line 12.

Algorithm 3. Tree edge weight increases

Input: MST $T = (V, E')$, tree edge (i, j) subject to a weight increase, positions s and f, weights w.

1: Assume j as child of i and perform `DFS`(j);
2: $k \leftarrow s$;
3: **while** $k < f$ **do**
4:    $(x, y) \leftarrow A(k)$;
5:    **if** `Label`$(x) \neq$ `Label`(y) **then**
6:       `Cut`(i, j);
7:       `Link`$(x, y, w(x, y))$;
8:       **return** updated minimum spanning tree T;
9:    **end if**
10:    $k \leftarrow k + 1$;
11: **end while**
12: **return** unchanged minimum spanning tree T;

Algorithm 3. can be adapted to make use of data structures based on the tree roots, such as ST-trees or RD-trees.

The edge list A, which is externally reordered, can be updated in $O(\log |E|)$ expected time if a skip list is used. The worst case occurs when edge (i, j) is

shifted from the first to the last position of A. In this case, $|E|$ edges may have to be considered for replacement. If DRD-trees (resp. ST-trees or RD-trees) are used, each `Label(v)` (resp. `Root(v)`) operation takes time $O(1)$ (resp. $O(\log|V|)$ or $O(|V|)$). Therefore, the overall complexity of Algorithm 3 is $O(|E|)$ if DRD-trees are used, $O(|E|\log|V|)$ if ST-trees are used, and $O(|E||V|)$ if RD-trees are used. The following theorem establishes the correctness of the approach:

Theorem 1. *Algorithms 1, 2, and 3 correctly update a minimum spanning tree.*

Proof. The structure of list A is such that if an edge (i, j) belongs to the current minimum spanning tree, then it is the first from left to right in the ordered list A connecting T_i and T_j. This is initially ensured by Kruskal's algorithm. It also holds for Algorithm 3, since by construction the latter selects the left-most minimum weight edge connecting T_i and T_j between positions s and f of list A.

We now show that letting the weight decreased edge be the right-most edge between those with the same weight preserves the structure of list A, ensuring the correctness of Algorithm 2. Assume edge (x, y) is the candidate edge to be replaced by (i, j) in the updated MST. If $w(x, y) > w(i, j)$, then edge (i, j) replaces (x, y) in the tree and becomes the left-most edge in list A connecting T_i and T_j, since otherwise (x, y) would not be in the current MST. In case $w(x, y) = w(i, j)$, then edge (i, j) does not replace (x, y). Due to the condition imposed by each reordering procedure, edge (i, j) is to the right of (x, y). Thus, (x, y) is the left-most in list A connecting T_i and T_j, and, therefore, the structure of list A associated with the updated minimum spanning tree is preserved. □

4 Computational Experiments

The computational experiments are presented in three sections. We first present experiments concerning the ability of DRD-trees to answer connectivity queries when compared to existing data structures. The next sections contain the experimental analysis of algorithms for updating minimum spanning trees of dynamic graphs, with numerical results for synthetic and realistic large graph instances.

The experiments were performed on a Pentium 4 processor with a 2.4 GHz clock and 768 Mbytes of RAM under GNU/Linux 2.6.16. The algorithms were coded in C++ and compiled with the GNU `g++` compiler version 4.1, using the optimization flag `-O2`. Although some codes were obtained from different authors, all algorithms and data structures were revised and optimized for this study. All processing times are average results over 100 instances of each size (ten different trees or graphs subject to ten different update sequences). Both random and structured sequences of updates are considered.

4.1 Dynamic Trees

We evaluate the behavior of DRD-trees regarding its efficiency to answer connectivity queries. Figure 1 depicts how fast the data structures can answer 100,000 connectivity queries in random trees containing from 2,000 to 400,000 vertices.

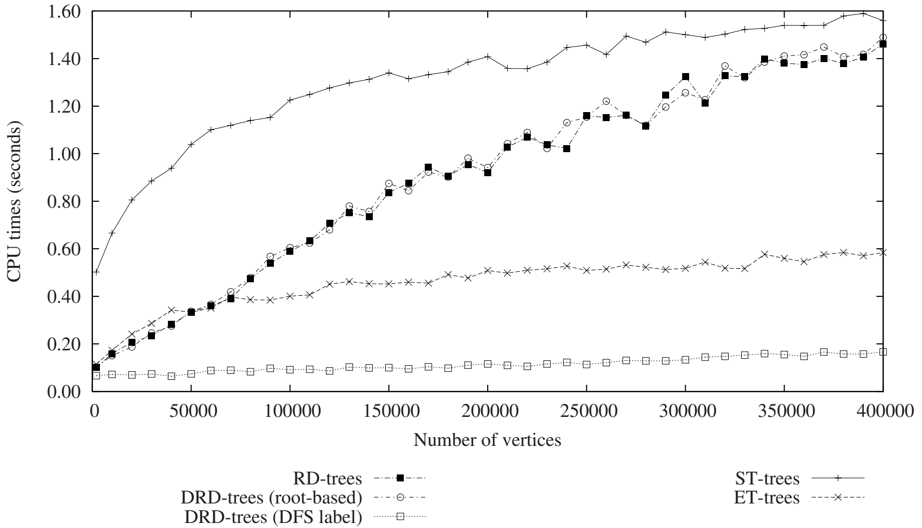


Fig. 1. CPU times for processing 100,000 connectivity queries in random trees

DRD-trees using the depth-first search labeling technique run in amortized constant time if the labels are used to process the connectivity queries, while RD-trees and DRD-trees run in linear time and ET-trees and ST-trees in logarithmic time (all of them using the root-based approach). DRD-trees are faster than the other data structures considered in this study.

4.2 Algorithms

This section presents a computational study addressing efficient algorithms for updating minimum spanning trees on dynamic graphs.

Cattaneo et al. [5] have shown that, except for very particular instances, a simple $O(m \log n)$-time algorithm has the best performance in practice and runs substantially faster than the poly-logarithmic algorithm HDT of Holm et al. [11]: the latter was faster than the first for only one out of five different classes of instances, namely those in k-clique graphs, in which small cliques are connected by some inter-clique edges and all updates involve only the inter-clique edges (see Section 4.2(a) below). Despite its theoretical running time of $O(\log^4 n)$ amortized, the complex chain of data structures supporting algorithm HDT does not seem to lead to fast implementations and may require too much memory. However, since Cattaneo et al. [5] have not applied it to grids and road networks, the performance of HDT on the instances with large memory requirements considered in Section 4.2(b) remains an open question.

We present the implemented algorithms together with their complexities in Table 1. The names of algorithms from [5] start by `C`, while those of variants of our approach start by `RT`. We indicate inside brackets the data structures used to maintain the minimum spanning tree. The following algorithms were compared:

C(ET+ST), C(DRD+ST), RT(DRD), RT(ET+ST), and RT(DRD+ST). Results for algorithms RT(RD) and RT(ST) are not reported, since their computation times are too large when compared to the others. Algorithms RT had the ordered list A implemented as a skip list with probability $p = 0.25$. The algorithms were applied to the same benchmark instances considered by other authors [5]. Processing times are average results over 100 instances of each size.

Table 1. Algorithms and running times per update

	Update type	
Algorithm	weight decreases	weight increases
RT(DRD)	$O(n)$	$O(m)$ amortized
RT(ET+ST)	$O(\log n)$	$O(m \log n)$
C(ET+ST)	$O(\log n)$	$O(m \log n)$
RT(DRD+ST)	$O(\log n)$	$O(m)$ amortized
C(DRD+ST)	$O(\log n)$	$O(m)$ amortized

(a) Synthetic Inputs: Results for random sequences of 20,000 updates applied to randomly generated graphs with 4,000 vertices and 8,000 to 100,000 edges are displayed in Figure 2. The results show that algorithm RT(DRD+ST) is slightly faster than the other approaches for random update sequences. Also, DRD-trees can be used to significantly improve the computation times of algorithm C(ET+ST).

As we are randomly selecting edges to update, the probability of increasing the weight of a tree edge is $(|V| - 1)/|E|$, decreasing with the increase of $|E|$ when $|V|$ is unchanged. Therefore, there are relatively fewer tree edge weights to be updated when the total number of edges increases and, consequently, the computation times become smaller. For all other cases (weight decreases and non-tree edge weight increases), we described algorithms that work in logarithmic time. The next experiments are more focused on the increase of tree edge weights, which can be more interesting than just randomly selecting any edge to update.

Results for structured sequences of 20,000 updates applied to randomly generated graphs with 4,000 vertices and 8,000 to 100,000 edges are shown in Figure 3. Here, 90% of the updates are tree edge weight increases and 10% are non-tree edge weight decreases. As predicted, the behavior in this case is the opposite to that in the previous situation. These are hard instances, since increasing the weight of tree edges and decreasing the weight of non-tree edges are the situations where Algorithms 2. and 3. are really called and used. In this context, the overall running time is dominated by the algorithms handling weight updates. The best options are variants RT(DRD) and RT(DRD+ST), since DRD-trees perform better when a large number of connectivity queries has to be processed. Moreover, algorithms C(DRD+ST) and C(ET+ST) did not perform well in this case, showing their inability to deal efficiently with hard update sequences.

Figure 4 presents results for graphs composed by isolated cliques connected by a few inter-clique edges, named k-clique graphs [5]. The updates are only incremental, applied exclusively to inter-clique edges. These are very hard instances,

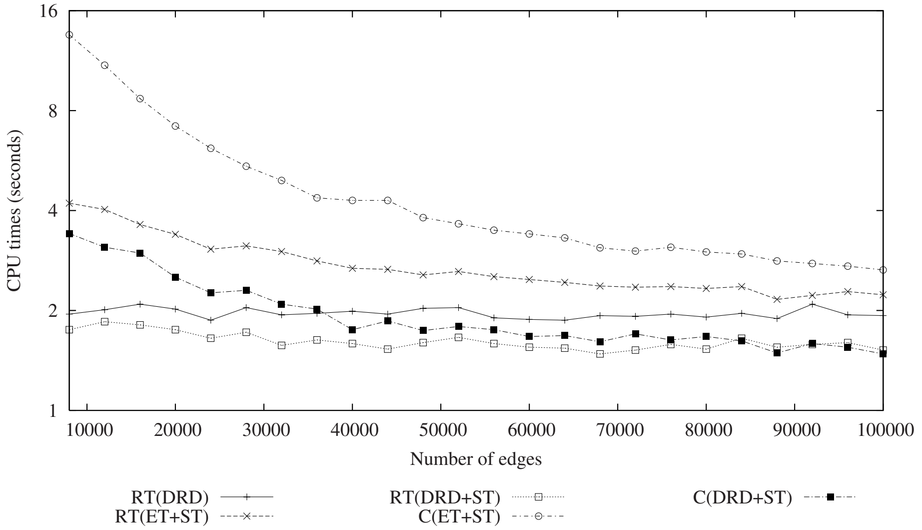


Fig. 2. CPU times for 20,000 random updates on randomly generated graphs

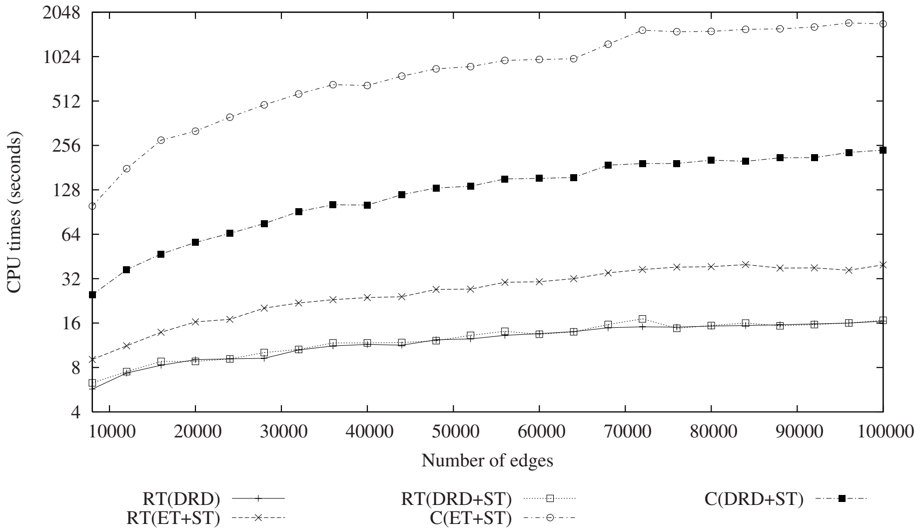


Fig. 3. CPU times for 20,000 structured updates on randomly generated graphs

since the set of candidates to replace the edge whose weight is increased is very small. The number of edges in this experiment ranges from 4,000 to 499,008, while the number of vertices is fixed at 2,000. The behavior of the algorithms is similar to that in Figure 3. The combination of edge weight increases with highly structured graphs resulted in the largest computation times among all

instances considered in this section. Variants `RT(DRD)` and `RT(DRD+ST)` are the best options in most cases. However, the latter is more robust, since Algorithm 2 has logarithmic complexity due to the use of ST-trees (as one may want to have good performance for both incremental and decremental updates).

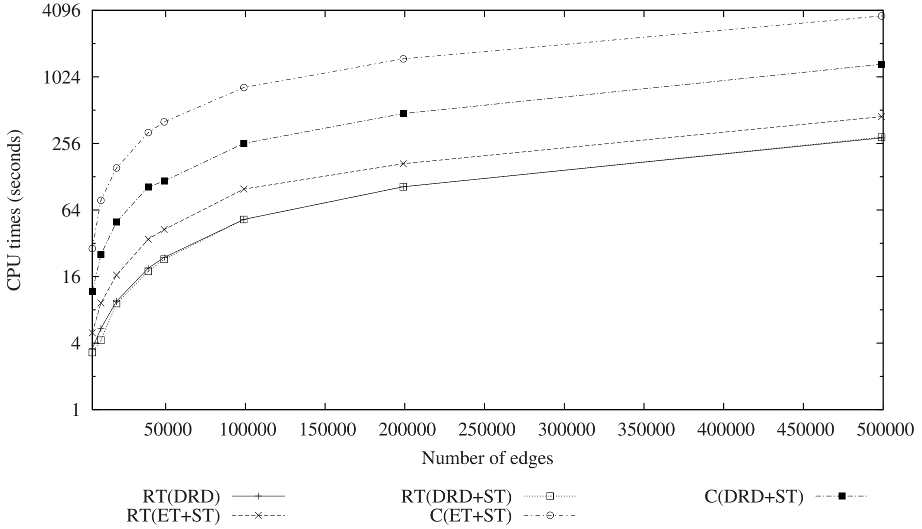


Fig. 4. CPU times for 20,000 incremental updates on k-clique instances

We conclude this section by reporting the number of connectivity queries performed by each approach – `C` and `RT` – on the k-clique instances corresponding to Figure 4. These numbers give an insight regarding the computation times obtained in all experiments above. Figure 5 shows that algorithms from [5] perform many more connectivity queries (per instance) than the proposed approach, what lead to the numerical advantage of algorithms `RT` along the experiments.

(b) Realistic Large Inputs: We now provide results for more realistic graphs from the DIMACS Implementation Challenge [6]. Table 2 displays results for four different types of graphs. Random4-n correspond to randomly generated graphs with $|E| = 4|V|$. Square-n graphs are generated in a two-dimensional square grid, with a small number of connections, while Long-n graphs are built on rectangular grids with long paths. Last, we present results for real-world instances named USA-road-d, derived from USA road networks. All update sequences are composed by 90% of weight increases and 10% of weight decreases.

The results in Table 2 show that variant `RT(DRD+ST)` performed better, providing fast algorithms for weight increases (using DRD-trees) and weight decreases (using ST-trees). This implementation was up to 110 times faster than `C(ET+ST)`, as observed for instance Random4-n.18.0. For road networks, variant `RT(DRD+ST)` also presented the best performance, achieving speedups of up to 51 times when compared to `C(ET+ST)`, as observed for instance USA-road-d.NY.

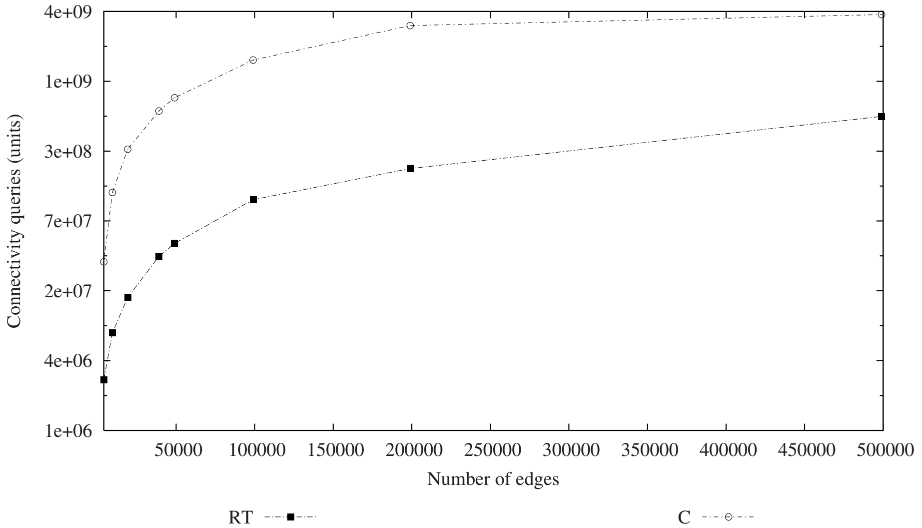


Fig. 5. Number of connectivity queries performed during the execution of 20,000 incremental updates on k-clique instances

Table 2. CPU times (seconds) for 20,000 structured updates on large instances

Instance	$\|V\|$	$\|E\|$	RT(DRD)	RT(ET+ST)	C(ET+ST)	RT(DRD+ST)	C(DRD+ST)
Random4-n.15.0	32,767	131,048	5.80	95.00	480.70	4.96	19.30
Random4-n.16.0	65,535	262,129	16.26	241.20	1,200.95	17.32	47.11
Random4-n.17.0	131,071	524,275	25.08	365.73	2,858.58	28.47	86.50
Random4-n.18.0	262,143	1,048,558	56.98	563.46	6,669.22	60.43	175.87
Random4-n.19.0	524,287	2,097,131	171.51	813.40	15,455.19	170.59	434.93
Square-n.15.0	32,760	65,156	4.87	32.60	141.13	4.19	10.33
Square-n.16.0	65,535	130,556	12.66	63.47	315.50	14.40	24.36
Square-n.17.0	131,043	261,360	28.23	101.89	847.71	30.42	51.03
Square-n.18.0	262,143	523,260	67.47	161.26	1,850.70	73.53	112.28
Square-n.19.0	524,175	1,046,900	132.73	324.63	4,321.13	140.09	229.80
Long-n.15.0	32,767	63,468	5.05	28.63	127.24	4.72	10.01
Long-n.16.0	65,535	126,956	15.03	45.25	313.92	16.17	26.86
Long-n.17.0	131,071	253,932	23.77	98.42	745.92	26.29	46.65
Long-n.18.0	262,143	507,884	59.35	184.20	1,683.73	62.95	103.20
Long-n.19.0	524,287	1,015,788	168.55	314.63	3,874.98	168.93	263.42
USA-road-d.NY	264,346	365,048	141.08	420.57	6,672.47	129.80	374.20
USA-road-d.BAY	321,270	397,414	188.44	674.85	6,832.83	174.84	416.60
USA-road-d.COL	435,666	521,199	233.69	889.54	7,351.65	216.42	560.23
USA-road-d.NW	1,207,945	1,410,384	589.47	3,196.67	23,843.47	566.80	1,518.80
USA-road-d.NE	1,524,453	1,934,008	752.20	4,112.73	47,990.40	737.60	2,152.56

(c) Dynamic vs. Non-Dynamic Algorithms: In this last section, we present results comparing the dynamic algorithms with the static ones. We considered the same instances used in the experiments reported in Table 2. In that

situation, algorithm `RT(DRD+ST)` processed 20,000 updates for each instance, using the amount of time showed in Table 2. We let Kruskal's and Prim's [13] algorithms run for the same time `RT(DRD+ST)` have run. On average, Prim's and Kruskal's algorithms were able to compute from scratch only 379 and 851 minimum spanning trees, respectively.

The dynamic approach was, in average, 52 (resp. 23) times faster than Prim's (resp. Kruskal's) algorithm. Thus, even though the proposed approach has the same theoretical complexity of other classical algorithms, these results emphasize the performance and the usefulness of dynamic algorithms.

5 Concluding Remarks

We proposed a new framework for the implementation of dynamic algorithms for updating the minimum spanning tree of a graph subject to edge weight changes. An extensive empirical analysis of different algorithms and variants has shown that the techniques presented in this paper are very suitable to hard update sequences and outperform the fastest algorithm in the literature.

We also proposed an easy-to-implement data structure for the linking and cutting trees problem. While DRD-trees provide linear time implementations for almost all operations, they are considerably faster when used to handle a large amount of connectivity queries. The experimental analysis showed that this structure not only reduced the computation times observed for the algorithm of Cattaneo et al. [5], but also contributed to the fastest algorithms in the computational experiments: `RT(DRD)` and `RT(DRD+ST)`.

Acknowledgments

We acknowledge Renato F. Werneck for making available his implementation of ST-trees and Giuseppe F. Italiano for his implementation of algorithm `C(ET+ST)`, both used in the computational experiments performed in this paper.

References

1. Amato, G., Cattaneo, G., Italiano, G.F.: Experimental analysis of dynamic minimum spanning tree algorithms (extended abstract). In: Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 314–323, New Orleans (1997)
2. Buriol, L.S., Resende, M.G.C., Ribeiro, C.C., Thorup, M.: A memetic algorithm for OSPF routing. In: Proceedings of the 6th INFORMS Telecom, pp. 187–188, Boca Raton (2002)
3. Buriol, L.S., Resende, M.G.C., Ribeiro, C.C., Thorup, M.: A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing. Networks 46, 36–56 (2005)
4. Buriol, L.S., Resende, M.G.C., Thorup, M.: Speeding up shortest path algorithms. Technical Report TD-5RJ8B, AT&T Labs Research (September 2003)

5. Cattaneo, G., Faruolo, P., Ferraro-Petrillo, U., Italiano, G.F.: Maintaining dynamic minimum spanning trees: An experimental study. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 111–125. Springer, Heidelberg (2002)
6. Demetrescu, C., Goldberg, A., Johnson, D.: Ninth DIMACS implementation challenge – shortest paths, 2006. On-line reference at http://www.dis.uniroma1.it/~challenge9/, last visited in June 23 (2006)
7. Eppstein, D., Galil, Z., Italiano, G.F., Nissemzweig, A.: Sparsification – A technique for speeding up dynamic graph algorithms. Journal of the ACM 44, 669–696 (1997)
8. Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. SIAM Journal on Computing 14, 781–798 (1985)
9. Frederickson, G.N.: Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In: Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, pp. 632–641, San Juan (1991)
10. Henzinger, M.H., King, V.: Maintaining minimum spanning trees in dynamic graphs. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 594–604. Springer, Heidelberg (1997)
11. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In: Proceedings of the 30th ACM Symposium on Theory of Computing, pp. 79–89, Dallas (1998)
12. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. In: Proc. of the American Mathematical Society, vol. 7, pp. 48–50 (1956)
13. Prim, R.C.: Shortest connection networks and some generalizations. Bell Systems Technical Journal 36, 1389–1401 (1957)
14. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM 33, 668–676 (1990)
15. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. Journal of Computer and System Sciences 26, 362–391 (1983)
16. Spira, P.M., Pan, A.: On finding and updating spanning trees and shortest paths. SIAM Journal on Computing 4, 375–380 (1975)
17. Werneck, R.F., Tarjan, R.E.: Self-adjusting top trees. In: Proc. of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 813–822, Vancouver (2005)
18. Zaroliagis, C.D.: Implementations and experimental studies of dynamic graph algorithms. In: Fleischer, R., Moret, B.M.E., Schmidt, E.M. (eds.) Experimental Algorithmics. LNCS, vol. 2547, pp. 229–278. Springer, Heidelberg (2002)

An Efficient Implementation 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 this paper, we present an approach, based on dynamic programming, for solving 0-1 multi-objective knapsack problems. The main idea of the approach relies on the use of several complementary dominance relations to discard partial solutions that cannot lead to new non-dominated criterion vectors. This way, we obtain an efficient method that outperforms the existing methods both in terms of CPU time and size of solved instances. Extensive numerical experiments on various types of instances are reported. A comparison with other exact methods is also performed. In addition, for the first time to our knowledge, we present experiments in the three-objective case.

Keywords: multi-objective knapsack problem, efficient solutions, dynamic programming, dominance relations, combinatorial optimization.

1 Introduction

In multi-objective combinatorial optimization, a major challenge is to develop efficient procedures to generate efficient solutions, that have the property that no improvement on any objective is possible without sacrificing on at least another objective. The aim is thus to find the efficient set (which consists of all the efficient solutions) or, more often, a reduced efficient set (which consists of only one solution for each non-dominated criterion vector). The reader can refer to [1] about multi-objective combinatorial optimization.

This paper deals with a particular multi-objective combinatorial optimization problem: the 0-1 multi-objective knapsack problem. The single-objective version of this problem has been studied extensively in the literature (see,*e.g*, [2,3]). Moreover, in the multi-objective case, many real-world applications are reported dealing with capital budgeting [4], relocation issues arising in conservation biology [5], and planning remediation of contaminated lightstation sites [6].

Several exact approaches have been proposed in the literature to find the efficient set or a reduced efficient set for the multi-objective knapsack problem. We first mention a theoretical work [7], without experimental results, where several dynamic programming formulations are presented. Two specific methods, with extensive experimental results, have been proposed: the two-phase method including a branch and bound algorithm proposed in [8], and the method of

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 406–419, 2007.
© Springer-Verlag Berlin Heidelberg 2007

[9], based on transformation of the problem into a bi-objective shortest path problem. All these methods have been designed for the bi-objective case and cannot be extended in a straightforward way to a higher number of objectives.

In this paper, we present a new approach based on dynamic programming. The main idea of the approach relies on the use of several complementary dominance relations to discard partial solutions that cannot lead to new non-dominated criterion vectors. This way, we obtain an efficient method that outperforms the existing methods both in terms of CPU time and size of solved instances (up to 4000 items in less than 2 hours in the bi-objective case). In our experiments, we compare our approach with the method of [9], which is the most efficient method currently known, and with an exact method based on a commercial Integer Programming solver. In addition, for the first time to our knowledge, we present experiments in the three-objective case.

This paper is organized as follows. In section 2, we review basic concepts about multi-objective optimization and formally define the multi-objective knapsack problem. Section 3 presents the dynamic programming approach and the dominance relations. Section 4 is devoted to implementation issues. Computational experiments and results are reported in section 5. Conclusions are provided in a final section. All proofs are given in the appendix section.

2 Preliminaries

2.1 Multi-objective Optimization

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*. Thus, the *efficient set* is defined as $E(X) = \{x \in X : \forall x' \in X,\ \text{not}(x' \Delta x)\}$. The set of non-dominated criterion vectors, which corresponds to the image of the efficient set in the criterion space, is denoted by ND. Since the efficient set can contain different solutions corresponding to the same criterion vector, any subset of $E(X)$ that contains one and only one solution for every non-dominated criterion vector is called a *reduced efficient set*. Observe that $X' \subseteq X$ is a reduced efficient set if and only if it is a *covering* and *independent set* with respect to $\underline{\Delta}$. We 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 $b, b' \in B,\ b \neq b',\ \text{not}(b \succsim b')$.

2.2 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 $v_1^k, \dots, v_p^k$ ($k = 1, \dots, n$). A feasible solution is represented by a vector $x = (x_1, \dots, x_n)$ 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 weight 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 $f_i(x) = \sum_{k=1}^{n} v_i^k x_k$ ($i = 1, \dots, p$). For any instance of this problem, we aim at determining the set of non-dominated criterion vectors.

3 Dynamic Programming and Dominance Relations

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 in DP (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 ($k = 1, \dots, n$). A state $s^k = (s_1^k, \dots, s_p^k, s_{p+1}^k) \in S^k$ represents a feasible solution of value s_i^k on the ith objective ($i = 1, \dots, p$) and of weight s_{p+1}^k. Thus, we have $S^k = S^{k-1} \cup \{(s_1^{k-1} + v_1^k, \dots, s_p^{k-1} + v_p^k, s_{p+1}^{k-1} + w^k) : s_{p+1}^{k-1} + w^k \leq W, s^{k-1} \in S^{k-1}\}$ for $k = 1, \dots, n$ where the initial set of states S^0 contains only the state $s^0 = (0, \dots, 0)$ corresponding to the empty knapsack. In the following, we identify a state and its corresponding feasible solution. Thus, relation $\underline{\Delta}$ defined on X is also valid on S^k, and we have $s^k \underline{\Delta} \tilde{s}^k$ if and only if $s_i^k \geq \tilde{s}_i^k$, $i = 1, \dots, p$.

Definition 1 (Completion, extension, restriction). *For any state $s^k \in S^k$ ($k < n$), a* completion *of s^k is any, possibly empty, subset $J \subseteq \{k+1, \dots, n\}$ 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 $i = 1, \dots, p$ 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 Dominance Relations in Dynamic Programming

The efficiency of DP depends crucially on the possibility of reducing the set of states at each phase. For this purpose, dominance relations between states are used to discard states at any phase. Dominance relations are defined as follows.

Definition 2 (Dominance relation between states). *A relation D^k on S^k, $k = 1, \dots, n$, is a dominance relation, if for all $s^k, \tilde{s}^k \in S^k$,*

$$s^k D^k \tilde{s}^k \Rightarrow \forall \tilde{s}^n \in Ext(\tilde{s}^k), \exists s^n \in Ext(s^k), s^n \underline{\Delta} \tilde{s}^n \quad (1)$$

A dominance relation D^k is not necessarily transitive. However, due to the transitivity of $\underline{\Delta}$, if D^k is a dominance relation then its transitive closure $\widehat{D}^k$ is also a dominance relation.

We introduce now a way of using dominance relations in Algorithm DP (see Algorithm 1). At each phase k, Algorithm DP 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 dominance relation D^k to each state of T^k 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).

Algorithm 1: Dynamic Programming

```
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          nonDominated ← true ; j ← 1;
7          while j ≤ ℓ_i and nonDominated do
8              if s̃^{k(j)} D^k s^{k(i)} then nonDominated ← false
9              else if s^{k(i)} D^k s̃^{k(j)} then C^k ← C^k \ {s̃^{k(j)}};
10             j ← j + 1;
11         while j ≤ ℓ_i do
12             if s^{k(i)} D^k s̃^{k(j)} then C^k ← C^k \ {s̃^{k(j)}};
13             j ← j + 1;
14         if nonDominated then C^k ← C^k ∪ {s^{k(i)}};
15 return C^n;
```

The following results characterize the set C^k obtained at the end of each phase k and establish the validity of Algorithm DP.

Proposition 1. *For any dominance relation D^k on S^k, the set C^k obtained at the end of phase k in Algorithm DP is a covering set of T^k with respect to $\widehat{D}^k$ that is also independent with respect to D^k ($k = 1, \dots, n$).*

Proof. Clearly, C^k is independent with respect to D^k, since we insert in C^k a state s^k at step 14 only if it is non-dominated by all others states of C^k (step 8) and we have removed from C^k all states dominated by s^k (step 9).

We have $\tilde{s}^k \in T^k \backslash C^k$ either because it did not pass the test at step 8 or was removed at step 9 or 12. In both cases, this is due to a state $\bar{s}^k$ already in C^k or to be included in C^k (at step 14) such that $\bar{s}^k \widehat{D}^k \tilde{s}^k$. Indeed, in the first case this is obvious since we have $\bar{s}^k D^k \tilde{s}^k$, and in the second case we can have either $\bar{s}^k D^k \tilde{s}^k$ or there exists a state $\bar{s}'^k$ such that $\bar{s}'^k D^k \tilde{s}^k$, that is not added to C^k (step 14) due to a state $\bar{s}^k$ currently in C^k (step 8) such that $\bar{s}^k D^k \bar{s}'^k D^k \tilde{s}^k$. In both cases, it may happen that $\bar{s}^k$ will be removed from C^k at a later iteration of the for loop (at step 9 or 12) if there exists a new state $\hat{s}^k \in T^k$, such that $\hat{s}^k D^k \bar{s}^k$. However, transitivity of $\widehat{D}^k$ ensures the existence, at the end of phase k, of a state $s^k \in C^k$ such that $s^k \widehat{D}^k \tilde{s}^k$. □

Theorem 1. *For any family of dominance relations $D^1, \dots, D^n$, Algorithm DP returns C^n a covering set of S^n with respect to $\underline{\Delta}$. Moreover, if $D^n = \underline{\Delta}$, C^n represents the set ND of non-dominated criterion vectors.*

Proof. Considering $\tilde{s}^n \in S^n \backslash C^n$, all its restrictions have been removed using D^k during phases $k \leq n$. Let k_1 be the highest phase where T^{k_1} still contains restrictions of $\tilde{s}^n$, which will be removed by applying D^{k_1}. Consider any of these restrictions, denoted by $\tilde{s}^{k_1}_{(n)}$. Since $\tilde{s}^{k_1}_{(n)} \in T^{k_1} \backslash C^{k_1}$, we know from Proposition 1, that there exists $s^{k_1} \in C^{k_1}$ such that $s^{k_1} \widehat{D}^{k_1} \tilde{s}^{k_1}_{(n)}$. Since $\widehat{D}^{k_1}$ is a dominance relation, by (1), we have that for all extensions of $\tilde{s}^{k_1}_{(n)}$, and in particular for $\tilde{s}^n$, there exists $s^{n_1} \in \mathrm{Ext}(s^{k_1})$ such that $s^{n_1} \underline{\Delta} \tilde{s}^n$. If $s^{n_1} \in C^n$, then the covering property holds. Otherwise, there exists a phase $k_2 > k_1$, corresponding to the highest phase where T^{k_2} still contains restrictions of s^{n_1}, which will be removed by applying D^{k_2}. Consider any of these restrictions, denoted by $s^{k_2}_{(n_1)}$. As before, we establish the existence of a state $s^{k_2} \in C^{k_2}$ such that there exists $s^{n_2} \in \mathrm{Ext}(s^{k_2})$ such that $s^{n_2} \underline{\Delta} s^{n_1}$. Transitivity of $\underline{\Delta}$ ensures that $s^{n_2} \underline{\Delta} \tilde{s}^n$. By repeating this process, we establish the existence of a state $s^n \in C^n$, such that $s^n \underline{\Delta} \tilde{s}^n$.

By Proposition 1, if $D^n = \underline{\Delta}$, C^n is an independent set with respect to $\underline{\Delta}$. Thus C^n, which corresponds to a reduced efficient set, represents the set of non dominated vectors. □

When dominance relation D^k is transitive, Algorithm DP can be drastically simplified in several ways. First, when we identify, at step 8, a state $\tilde{s}^{k(j)} \in C^k$ such that $\tilde{s}^{k(j)} D^k s^{k(i)}$, transitivity of D^k and independence of C^k with respect to D^k ensure that $s^{k(i)}$ cannot dominate any state in C^k, which makes the loop 11-13 useless. Second, if we identify, at step 9, a state $\tilde{s}^{k(j)} \in C^k$ such that $s^{k(i)} D^k \tilde{s}^{k(j)}$, transitivity of D^k and independence of C^k with respect to D^k ensure that $s^{k(i)}$ cannot be dominated by a state of C^k, which allows us to leave immediately the current loop 7-10.

Further improvements can still be made since it is usually possible to generate states of $T^k = \{s^{k(1)}, \dots, s^{k(r)}\}$ according to a *dominance preserving order* for D^k such that for all $i < j$ $(1 \leq i,j \leq r)$ we have either not$(s^{k(j)} D^k s^{k(i)})$ or $\left(s^{k(j)} D^k s^{k(i)} \text{ and } s^{k(i)} D^k s^{k(j)}\right)$. The following proposition gives a necessary and sufficient condition to establish the existence of a dominance preserving order for a dominance relation.

Proposition 2. *Let D^k be a dominance relation on S^k. There exists a dominance preserving order for D^k if and only if D^k does not admit cycles in its asymmetric part.*

Proof. $\Rightarrow$ The existence of a cycle in the asymmetric part of D^k would imply the existence of two consecutive states $s^{k(j)}$ and $s^{k(i)}$ on this cycle with $j > i$, a contradiction.

$\Leftarrow$ Any topological order based on the asymmetric part of D^k is a dominance preserving order for D^k. □

If states of T^k are generated according to a dominance preserving order for D^k, step 9 and loop 11-13 can be omitted.

In our presentation, Algorithm DP provides us with the set of non-dominated criterion vectors. The approach can be easily adapted to obtain a reduced efficient set by adding to each generated state components characterizing its corresponding solution. Moreover, the efficient set can be obtained by using dominance relations D^k $(k = 1, \ldots, n)$ satisfying condition (1), where $\underline{\Delta}$ is replaced by Δ, and provided that C^n is an independent set with respect to Δ.

4 Implementation Issues

We first present the order in which we consider items in the sequential process (section 4.1). Then, we present three dominance relations that we use in DP (section 4.2) and a brief explanation of the way of applying them (section 4.3).

4.1 Item Order

The order in which items are considered is a crucial implementation issue in DP. In the single-objective knapsack problem, it is well-known that, in order to obtain a good solution, items should usually be considered in decreasing order of value to weight ratios v^k/w^k (assuming that ties are solved arbitrarily) [2,3]. For the multi-objective version, there is no such a natural order.

We introduce now three orders $\mathcal{O}^{sum}$, $\mathcal{O}^{\max}$, and $\mathcal{O}^{\min}$ that are derived by aggregating orders $\mathcal{O}^i$ induced by the ratios v_i^k/w^k for each criterion $(i = 1, \ldots, p)$. Let r_i^ℓ be the rank or position of item ℓ in order $\mathcal{O}^i$. $\mathcal{O}^{sum}$ denotes an order according to increasing values of the sum of the ranks of items in the p orders $\mathcal{O}^i$ $(i = 1, \ldots, p)$. $\mathcal{O}^{\max}$ denotes an order according to the increasing values of the maximum or worst rank of items in the p orders $\mathcal{O}^i$ $(i = 1, \ldots, p)$, where the worst rank of item ℓ in the p orders $\mathcal{O}^i$ $(i = 1, \ldots, p)$ is computed by $\max_{i=1,\ldots,p}\{r_i^\ell\} + \frac{1}{pn}\sum_{i=1}^{p} r_i^\ell$ in order to discriminate items with the same maximum rank. Similarly, $\mathcal{O}^{\min}$ denotes an order according to the increasing values of the minimum rank of items in the p orders $\mathcal{O}^i$ $(i = 1, \ldots, p)$.

In the computational experiments, in Section 5, we show the impact of the order on the efficiency of our approach.

4.2 Dominance Relations

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 the three dominance relations used in our method. The first two relations are very easy to establish and the last one, although more difficult to establish, is considered owing to its complementarity with the two others.

We first present a dominance relation 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:

$$\text{for all } s^k, \tilde{s}^k \in S^k,\ s^k D_r^k \tilde{s}^k \Leftrightarrow \begin{cases} \tilde{s}^k \in S^{k-1}, \\ s^k = (\tilde{s}_1^k + v_1^k, \ldots, \tilde{s}_p^k + v_p^k, \tilde{s}_{p+1}^k + w^k) \\ \tilde{s}_{p+1}^k \leq W - \sum_{j=k}^n w^j \end{cases}$$

Proposition 3 (Relation D_r^k)
(a) D_r^k is a dominance relation
(b) D_r^k is transitive
(c) D_r^k admits dominance preserving orders

Proof (a) Consider two states s^k and $\tilde{s}^k$ such that $s^k D_r^k \tilde{s}^k$. This implies, that $s^k \underline{\Delta} \tilde{s}^k$. Moreover, since $s_{p+1}^k = \tilde{s}_{p+1}^k + w^k \leq W - \sum_{j=k+1}^n w^j$, any subset $J \subseteq \{k+1, \ldots, n\}$ is a completion for $\tilde{s}^k$ and s^k. Thus, for all $\tilde{s}^n \in \mathrm{Ext}(\tilde{s}^k)$, there exists $s^n \in \mathrm{Ext}(s^k)$, based on the same completion as $\tilde{s}^n$, such that $s^n \underline{\Delta} \tilde{s}^n$. This establishes that D_r^k satisfies condition (1) of Definition 2.
(b) Obvious.
(c) By Proposition 2, since D_r^k is transitive. □

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.

A second dominance relation $D_{\underline{\Delta}}^k$ is defined on S^k for $k = 1, \ldots, n$ by:

$$\text{for all } s^k, \tilde{s}^k \in S^k,\ s^k D_{\underline{\Delta}}^k \tilde{s}^k \Leftrightarrow \begin{cases} s^k \underline{\Delta} \tilde{s}^k \\ s_{p+1}^k \leq \tilde{s}_{p+1}^k, \text{ if } k < n \end{cases}$$

Dominance relation $D_{\underline{\Delta}}^k$ is a generalization to the multi-objective case of the dominance relation usually attributed to Weingartner and Ness [10] and used in the classical Nemhauser and Ullmann algorithm [11].

Proposition 4 (Relation $D^k_{\underline{\Delta}}$)

(a) $D^k_{\underline{\Delta}}$ is a dominance relation
(b) $D^k_{\underline{\Delta}}$ is transitive
(c) $D^k_{\underline{\Delta}}$ admits dominance preserving orders
(d) $D^n_{\underline{\Delta}} = \underline{\Delta}$

Proof (a) Consider two states s^k and $\tilde{s}^k$ such that $s^k D^k_{\underline{\Delta}} \tilde{s}^k$. This implies, that $s^k \underline{\Delta} \tilde{s}^k$. Moreover, since $s^k_{p+1} \leq \tilde{s}^k_{p+1}$, any subset $J \subseteq \{k+1, \ldots, n\}$ that is a completion for $\tilde{s}^k$ is also a completion for s^k. Thus, for all $\tilde{s}^n \in \mathrm{Ext}(\tilde{s}^n)$, there exists $s^n \in \mathrm{Ext}(s^n)$, based on the same completion as $\tilde{s}^n$, such that $s^n \underline{\Delta} \tilde{s}^n$. This establishes that $D^k_{\underline{\Delta}}$ satisfies condition (1) of Definition 2.
(b) Obvious.
(c) By Proposition 2, since $D^k_{\underline{\Delta}}$ is transitive.
(d) By definition. □

Relation $D^k_{\underline{\Delta}}$ 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.

The third dominance relation is based on the comparison between extensions of a state and an upper bound of 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 $s^n \in \mathrm{Ext}(s^k)$ we have $u_i \geq s^n_i$, $i = 1, \ldots, p$.

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 $s^k_i + \sum_{j \in J} v^j_i \geq \tilde{u}_i$, $i = 1, \ldots, p$, 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. After relabeling items $k+1, \ldots, n$ according to order $\mathcal{O}^{sum}$, completion J' is obtained by inserting sequentially the remaining items into the solution provided that the capacity constraint is respected. More precisely, J' correspond to J_n where $J_k = \emptyset$ and $J_\ell = J_{\ell-1} \cup \{\ell\}$ if $s^k_{p+1} + \sum_{j \in J_{\ell-1}} w^j + w^\ell \leq W$, $\ell = k+1, \ldots, n$. J'' is defined similarly by relabeling items according to order $\mathcal{O}^{\max}$.

To compute u, we use the upper bound presented in [2] for each criterion value. Let us first define $\overline{W}(s^k) = W - s^k_{p+1}$ the residual capacity associated to state $s^k \in S^k$. We denote by $c_i = \min\{\ell_i \in \{k+1, \ldots, n\} : \sum_{j=k+1}^{\ell_i} w^j > \overline{W}(s^k)\}$ the position of the first item that cannot be added to state $s^k \in S^k$ when items $k+1, \ldots, n$ are relabeled according to order $\mathcal{O}^i$. Thus, according to [2, Th 2.2], when items $k+1, \ldots, n$ are relabeled according to order $\mathcal{O}^i$, an upper bound on the ith criterion value of $s^k \in S^k$ is for $i = 1, \ldots, p$:

$$u_i = s^k_i + \sum_{j=k+1}^{c_i - 1} v^j_i + \max\left\{ \left\lfloor \overline{W}(s^k) \frac{v_i^{c_i+1}}{w^{c_i+1}} \right\rfloor, \left\lfloor v_i^{c_i} - (w^{c_i} - \overline{W}(s^k)) \frac{v_i^{c_i-1}}{w^{c_i-1}} \right\rfloor \right\} \quad (2)$$

Finally, we define D^k_b a particular dominance relation of this general type for $k = 1, \ldots, n$ by:

$$\text{for all } s^k, \tilde{s}^k \in S^k,\ s^k D_b^k \tilde{s}^k \Leftrightarrow \begin{cases} s_i^k + \sum_{j \in J'} v_i^j \geq \tilde{u}_i, & i = 1, \ldots, p \\ \text{or} \\ s_i^k + \sum_{j \in J''} v_i^j \geq \tilde{u}_i, & i = 1, \ldots, p \end{cases}$$

where $\tilde{u} = (\tilde{u}_1, \ldots, \tilde{u}_p)$ is the upper bound for $\tilde{s}^k$ computed according to (2).

Proposition 5 (Relation D_b^k)
(a) D_b^k is a dominance relation
(b) D_b^k is transitive
(c) D_b^k admits dominance preserving orders
(d) $D_b^n = \underline{\Delta}$

Proof (a) Consider states s^k and $\tilde{s}^k$ such that $s^k D_b^k \tilde{s}^k$. This implies that there exists $J \in \{J', J''\}$ leading to an extension s^n of s^k such that $s^n \underline{\Delta} \tilde{u}$. Moreover, since $\tilde{u}$ is an upper bound of $\tilde{s}^k$, we have $\tilde{u} \underline{\Delta} \tilde{s}^n$, for all $\tilde{s}^n \in \text{Ext}(\tilde{s}^k)$. Thus, by transitivity of $\underline{\Delta}$, we get $s^n \underline{\Delta} \tilde{s}^n$, which establishes that D_b^k satisfies condition (1) of Definition 2.
(b) Consider states s^k, $\tilde{s}^k$, and $\bar{s}^k$ such that $s^k D_b^k \tilde{s}^k$ and $\tilde{s}^k D_b^k \bar{s}^k$. This implies that, on the one hand, there exists $J_1 \in \{J', J''\}$ such that $s_i^k + \sum_{j \in J_1} v_i^j \geq \tilde{u}_i$ $(i = 1, \ldots, p)$, and on the other hand, there exists $J_2 \in \{J', J''\}$ such that $\tilde{s}_i^k + \sum_{j \in J_2} v_i^j \geq \bar{u}_i$ $(i = 1, \ldots, p)$. Since $\tilde{u}$ is an upper bound for $\tilde{s}^k$ we have $\tilde{u}_i \geq \tilde{s}_i^k + \sum_{j \in J_2} v_i^j$ $(i = 1, \ldots, p)$. Thus we get $s^k D_b^k \bar{s}^k$.
(c) By Proposition 2, since D_b^k is transitive.
(d) By definition. □

D_b^k is harder to check than relations D_r^k and $D_{\underline{\Delta}}^k$ since it requires much more tests and state-dependent information.

Obviously, relation D_b^k would have been richer if we had used additional completions (according to other orders) for s^k and computed instead of one upper bound u, an upper bound set using, *e.g.*, the techniques presented in [12]. Nevertheless, in our context since we have to check D_b^k for many states, enriching D_b^k in this way would be extremely time consuming.

4.3 Use of Multiple Dominance Relations

In order to be efficient, we will use the three dominance relations presented in section 4.2 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 $D_{\underline{\Delta}}^k$) and then test on a reduced set of states dominance relations requiring a larger computation time (such as D_b^k).

The running time of Algorithm DP using these relations is in $O(n(\min\{W, U\} U^{p-1})^2)$ where U is an upper bound on the value of all solutions on all criteria, since C^k, which contains only non-dominated vectors with respect to profit values and weight, has a cardinality in $O(\min\{W, U\}U^{p-1})$. Based on ideas of [13], in the bi-objective case, in order to remove efficiently dominated states at each phase,

we use an AVL tree [14, sec. 6.3.3] for storing states which leads to a significant improvement of the running time to $O(n \min\{W, U\} \log(\min\{W, U\}))$. Observe that space complexity of Algorithm DP is in $O(\min\{W, U\}U^{p-1})$.

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, $v_k^1 \in_R [1, 1000]$, $v_k^2 \in_R [1, 1000]$ and $w_k \in_R [1, 1000]$

B: Unconflicting instances, where v_k^1 is correlated with v_k^2, *i.e.* $v_k^1 \in_R [111, 1000]$ and $v_k^2 \in_R [v_k^1 - 100, v_k^1 + 100]$, and $w_k \in_R [1, 1000]$

C: Uncorrelated conflicting instances, where v_k^1 and v_k^2 are mirror values, *i.e.* $v_k^1 \in_R [1, 1000]$, $v_k^2 \in_R [\max\{900 - v_1^k; 1\}, \min\{1100 - v_1^k; 1000\}]$, and $w_k \in_R [1, 1000]$

D: Correlated conflicting instances, where v_k^1 and v_2^k are mirror values, and w_k is correlated with v_k^1 and v_2^k, *i.e.* $v_k^1 \in_R [1, 1000]$, $v_k^2 \in_R [\max\{900 - v_1^k; 1\}, \min\{1100 - v_1^k; 1000\}]$, 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 mono-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. For instances of types C and D, items are around the line $y = -x + 1000$. In instances of type D, w^k is correlated with v_k^1, v_k^2. These instances were introduced in order to verify if correlated instances are harder than uncorrelated instances as in the single-criterion context [2].

For three-objective experiments, we considered the generalization of random instances of type A where $v_k^i \in_R [1, 1000]$ for $i = 1, \ldots, 3$ and $w_k \in_R [1, 1000]$.

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.

5.2 Results in the Bi-objective Case

First, in the experiments, we try to determine the best order to sort items for DP. Table 1 shows clearly that the way of ordering items has a dramatic impact on the

Table 1. Impact of different orders of items in our approach (Average CPU time in seconds, $p = 2$)

Type	n	$\mathcal{O}^{\max}$	$\mathcal{O}^{sum}$	$\mathcal{O}^{\min}$	Random
A	300	84.001	100.280	94.598	178.722
B	600	1.141	1.084	1.403	77.699
C	200	59.986	60.061	85.851	107.973
D	90	20.795	23.687	35.426	31.659

CPU time and that order $\mathcal{O}^{\max}$ is significantly better for all types of instances. Thus, in the following, items are sorted and labeled according to $\mathcal{O}^{\max}$.

Second, we show the complementarity of dominance relations D^k_r, $D^k_{\underline{\Delta}}$, and D^k_b. Table 2 establishes that it is always better to use the three relations, due to their complementarity.

Table 2. Complementarity of dominance relations D^k_r, $D^k_{\underline{\Delta}}$, and D^k_b in our approach (Average CPU time in seconds, $p = 2$)

Type	n	$D^k_{\underline{\Delta}}$	D^k_r and $D^k_{\underline{\Delta}}$	$D^k_{\underline{\Delta}}$ and D^k_b	D^k_r, $D^k_{\underline{\Delta}}$, and D^k_b
A	300	272.628	157.139	85.076	84.001
B	600	230.908	174.015	1.188	1.141
C	200	122.706	63.557	61.696	59.986
D	90	46.137	24.314	23.820	20.795

Table 3. Results of our approach on large size instances ($p = 2$)

Type	n	Time in (s)			$\|ND\|$			Avg
		Min	Avg	Max	Min	Avg	Max	$\max_k\{\|C^k\|\}$
A	100	0.152	0.328	0.600	98	159.3	251	17134.7
	300	57.475	84.001	101.354	905	1130.7	1651	898524.7
	500	677.398	889.347	1198.190	2034	2537.5	2997	5120514.7
	700	4046.450	5447.921	7250.530	3768	4814.8	5939	18959181.7
B	1000	4.328	8.812	15.100	105	157.0	218	134107.2
	2000	139.836	251.056	394.104	333	477.7	630	1595436.1
	3000	1192.190	1624.517	2180.860	800	966.9	1140	6578947.2
	4000	4172.530	6773.264	8328.280	1304	1542.3	1752	18642759.0
C	100	1.564	2.869	4.636	406	558.2	737	103921.5
	300	311.995	373.097	470.429	2510	2893.6	3297	3481238.4
	500	2433.320	4547.978	6481.970	5111	7112.1	9029	21282280.5
D	100	36.450	40.866	54.267	1591	1765.4	2030	1129490.3
	150	235.634	265.058	338.121	2985	3418.5	3892	4274973.9
	200	974.528	1145.922	1497.700	4862	5464.0	6639	12450615.5
	250	2798.040	3383.545	3871.240	7245	8154.7	8742	26999714.8

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 4000 items and the instances with the largest number of non-dominated criterion vectors are those of type D with 250 items for which the cardinality of the set of non-dominated criterion vectors is in average of 8154.7. As predicted, instances of type B are quasi mono-objective instances and have very few non-dominated criterion vectors. 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 250 items.

5.3 Comparison with Other Exact Methods in the Bi-objective Case

The results of a comparative study, in the bi-objective case, between the exact method of [9], an exact method based on a commercial Integer Programming (IP) solver and our approach using D_r^k, $D_{\underline{\Delta}}^k$, and D_b^k are presented in Table 4. We have selected the method of [9] since it is the most efficient method currently known. An exact method based on a commercial IP solver has been selected, on one hand, because it is relatively easy to implement, and on the other hand, since each efficient solution is found by solving only one linear program, this method has much less storage problems than the two others.

An exact method based on a commercial IP solver is presented in Algorithm 2. This algorithm relies on the idea that since the decision space $Z = \{f(x) : x \in X\}$ is included in $\mathbb{N}^2$, all efficient solutions can be enumerated in decreasing order of value on the first criterion. Cplex 9.0 is used as IP solver in Algorithm 2 which is written in C++.

Algorithm 2: Computing a reduced efficient set with an IP Solver

1. Generate y an optimal solution of $\max_{x \in X} f_1(x)$ and z an optimal solution of $\max_{x \in X} f_2(x)$;
2. Generate x^1 an optimal solution of $\max\{f_2(x) : x \in X, f_1(x) \geq f_1(y)\}$;
3. $X^\star \leftarrow X^\star \cup \{x^1\}$; $j \leftarrow 1$;
4. **while** $f_2(x^j) < f_2(z)$ **do**
5. $\quad \alpha \leftarrow f_2(z) - f_2(x^j) - 1$;
6. $\quad$ Generate x^{j+1} an optimal solution of $\max\{\alpha f_1(x) + f_2(x) : x \in X, f_2(x) \geq f_2(x^j) + 1\}$;
7. $\quad X^\star \leftarrow X^\star \cup \{x^{j+1}\}$; $j \leftarrow j + 1$;
8. **return** $X^\star$;

Table 4. Comparison between the exact method of [9], Algorithm 2 using Cplex and our approach

Type	n	Avg time in (s) [9]	Avg time in (s) Cplex	Avg time in (s) Our approach	Avg $\lvert ND \rvert$
A	100	2.476	5.343	0.328	159.3
	200	37.745	57.722	12.065	529.0
	300	163.787	285.406	84.001	1130.7
B	600	27.694	27.543	1.141	74.3
	700	47.527	29.701	2.299	78.6
	800	75.384	68.453	5.280	118.1
C	100	12.763	208.936	2.869	558.2
D	100	127.911	23126.926	40.866	1765.4

The three methods have been used on the same instances and the same computer. For the exact method of [9], we used the source code, in C, obtained from the authors. Table 4 presents results, in the bi-objective case, for instances of

type A, B, C, and D for increasing size of n while the method of [9] can solve all instances of the series considered. Since the method of [9] is very storage consuming, it can only solve instances of type A up to 300 items, of type B up to 800 items, of type C up to 100 items and of type D up to 100 items whereas we recall (see Table 3) that our approach can solve instances respectively up to 700, 4000, 500 and 250 items.

Considering CPU time, we can conclude that our approach is always faster than the exact method of [9] and than Algorithm 2 with Cplex on the considered instances. We can also observe that the CPU time needed to solve correlated and conflicting instances of type D by Algorithm 2 with Cplex is especially large (about 6.5 hours in average for instances 2D100). In addition, we can remark that the exact method of [9] cannot solve conflicting instances (type C and D) of moderate and large size for which the number of non-dominated criterion vectors is large. Indeed, the exact method of [9] does not work very well on instances with many non-dominated criterion vectors due to storage limitations.

5.4 Results in the Three-Objective Case

In table 5, we present results of our approach concerning large size instances of type A in the three-objective case. Observe that the number of non-dominated criterion vectors varies a lot. This explains the variation of the CPU time which is strongly related with the number of non-dominated criterion vectors.

Table 5. Results of our approach on instances of type A in the three-objective case

n	Time in (s)			$\|ND\|$			Avg
	Min	Avg	Max	Min	Avg	Max	$\max_k\{\|C^k\|\}$
10	0.000	0.000	0.000	4	8.3	18	20.9
30	0.000	0.012	0.028	31	112.9	193	1213.2
50	0.112	0.611	1.436	266	540.6	930	12146.5
70	4.204	16.837	44.858	810	1384.4	2145	64535.4
90	80.469	538.768	2236.230	2503	4020.3	6770	285252.1
110	273.597	3326.587	11572.700	3265	6398.3	9394	601784.6

6 Conclusions

The goal of this work has been to develop and experiment a new dynamic programming algorithm to solve the $0-1$ multi-objective knapsack problem. We showed that by using several complementary dominance relations, we obtain a method which outperforms experimentally the existing methods. In addition, our method is extremely efficient with regard to the other methods on the conflicting instances that model real world applications. Lastly, this method is the first one to our knowledge that can be applied for knapsack with more than two objectives and the results in the three-objective case are very satisfactory.

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 other multi-objective problems such as the multi-objective shortest path problem or multi-objective scheduling problems.

References

1. Ehrgott, M.: Multicriteria optimization. LNEMS, vol. 491. Springer, Heidelberg (2005)
2. Martello, S., Toth, P.: Knapsack Problems. Wiley, New York (1990)
3. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Heidelberg (2004)
4. Rosenblatt, M.J., Sinuany-Stern, Z.: Generating the discrete efficient frontier to the capital budgeting problem. Operations Research 37(3), 384–394 (1989)
5. Kostreva, M.M., Ogryczak, W., Tonkyn, D.W.: Relocation problems arising in conservation biology. Comp. and Math. with App. 37, 135–150 (1999)
6. Jenkins, L.: A bicriteria knapsack program for planning remediation of contaminated lightstation sites. Eur. J. Oper. Res. 140, 427–433 (2002)
7. Klamroth, K., Wiecek, M.: Dynamic programming approaches to the multiple criteria knapsack problem. Naval Research Logistics 47(1), 57–76 (2000)
8. Visée, M., Teghem, J., Pirlot, M., Ulungu, E.: Two-phases method and branch and bound procedures to solve the bi-objective knapsack problem. Journal of Global Optimization 12, 139–155 (1998)
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, 1865–1886 (2003)
10. Weignartner, H., Ness, D.: Methods for the solution of the multi-dimensional 0/1 knapsack problem. Operations Research 15(1), 83–103 (1967)
11. Nemhauser, G., Ullmann, Z.: Discrete dynamic programming and capital allocation. Management Science 15(9), 494–505 (1969)
12. Ehrgott, M., Gandibleux, X.: Bound sets for biobjective combinatorial optimization problems. To appear in Computers and Operations Research (2007)
13. Kung, H., Luccio, F., Preparata, F.: On finding the maxima of set of vectors. J. Assoc. Comput. Mach. 22(4), 469–476 (1975)
14. Knuth, D.E.: The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1997)

Trunk Packing Revisited

Ernst Althaus, Tobias Baumann, Elmar Schömer, and Kai Werth

Johannes-Gutenberg-Universität Mainz
Institut für Informatik
{althaus,tba,schoemer,werth}@informatik.uni-mainz.de

Abstract. For trunk packing problems only few approximation schemes are known, mostly designed for the European standard DIN 70020 [6] with equally sized boxes [8, 9, 11, 12]. In this paper two discretized approaches for the US standard SAE J1100 [10] are presented, which make use of different box sizes. An exact branch-and-bound algorithm for weighted independent sets on graphs is given, using the special structure of the SAE standard. Another branch-and-bound packing algorithm using linear programs is presented. With these algorithms axis-oriented packings of different box sizes in an arbitrary trunk geometry can be computed efficiently.

Keywords: approximation algorithms, branch and bound algorithms, branch and cut algorithms, graph algorithms, linear programming, packing problems, weighted independent set.

1 Introduction

The cooperation with a German car manufacturer has led to several efficient approximation schemes for the trunk packing problem according to the German standard DIN 70020 [4, 5, 6, 8, 11, 12]. This cooperation has been continued by exploring the baggage volume capacity according to the US standard SAE J1100 [10].

The computation of the baggage capacity has a significant influence on the car design process. According to international regulations, the baggage capacity is not equal to the continuous volume of a trunk which can be easily computed using a standard CAD program. The German standard DIN 70020 uses boxes of size $20 \times 10 \times 5$ cm, which equals 1 l per box. The American standard SAE J1100 uses a more realistic set of boxes: seven different box sizes are used, making an equivalent from 6 up to 67 litres. To create a valid[1] SAE J1100 packing, a few rules must be obeyed: The smallest boxes may not be used until none of the larger boxes fit into the trunk, and each box type has got a fixed number of instances to be used at maximum.

The trunk packing problem has recently been explored for the DIN 70020 case. In [5, 11], a discrete approach is followed using a discretization of the trunk volume into cubes of fixed size. Using discrete coordinates derived from these cubes as representatives for boxes, a conflict graph can be created and

[1] A packing is considered valid if it contains only boxes of allowed sizes and each box size does not occur more often than allowed.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 420–432, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the INDEPENDENT SET (IS) problem can be solved on this graph. [5, 11] also provide several algorithms using some structural properties of the conflict graph and optimization techniques for the underlying grid.

Cagan and Ding [3] presented a packing algorithm for SAE J1100 using extended pattern search. Additionally to this approach one could also make use of a conflict graph. The SAE standard differs from the DIN standard in two issues: At first, the goal is to find an independent set with the maximal covered volume. This problem is known as WEIGHTED INDEPENDENT SET (WIS). Second, the maximal WIS might not be a valid solution for the SAE J1100 standard which allows only a fixed number of occurences for each box type. Table 1 shows the allowed box types and their sizes. There also exists an irregular shaped golf bag (type G) which is currently not used.

For the WIS problem itself a greedy algorithm has been explored recently [7, 13]. An approximation ratio of $\min((\bar{d}_w+1)/2, (\delta_w+1)/2)$ is derived for the greedy algorithm, where $\bar{d}_w$ denotes the *weighted average degree*, and δ_w is the *weighted inductiveness* of a weighted undirected graph as defined in [13]. The special structure of the conflict graph, however, causes that this approximation ratio can be achieved easily and provides rather poor values for the trunk packing problem.

Some branch-and-bound algorithms for the WIS problem on general graphs have been evaluated [15]. An approximation algorithm using local search has been developed by Berman [2]. This algorithm achieves an approximation ratio of $d/2$ in a d-claw free graph. A d-claw is an induced subgraph containing an independent set of d nodes, and a center node that is connected to all members of the independent set. A graph is called d-claw free if it does not contain any d-claws. Unfortunately, the conflict graph as constructed in section 2 contains d-claws for very large d, so this approximation ratio would not provide any gain for the trunk packing problem. Furthermore, these algorithms apply to graphs with arbitrary integer weights. In the trunk packing case, the fact that only seven different weights are present can be exploited. Thus it is necessary to find other efficient algorithms for this special version of the WIS problem.

Additionally, a continuous approach was developed using Simulated Annealing techniques (see [4, 8, 12]). Using moves like *translation*, *rotation*, *creation* and *deletion*, the discretized solutions could be improved in many cases. A new promising approach for continuous packing is the simulation of the physical

Table 1. Allowed box types for the SAE J1100 trunk packing problem

box	max.	inch (″)			mm			Volume	
type	occurrences	l	w	h	l	w	h	l	ft^3
A	4	24	19	9	610	483	229	67.47	2.375
B	4	18	13	6.5	457	330	165	24.88	0.880
C	2	26	16	9	660	406	229	61.36	2.167
D	2	21	18	8.5	533	457	216	52.61	1.859
E	2	15	9	8	381	229	203	17.71	0.625
F	2	21	14	7	533	356	178	33.78	1.191
H	20	12.8	6	4.5	325	152	114	5.63	0.200

processes during the motion of rigid bodies. There it is possible, by pushing boxes, to create free space at a designated place and therefore to insert new boxes [1]. The idea to use a contact simulation for a packing problem arose from a programming contest about Circle Packing [17].

This paper is structured as follows: Section 2 deals with the WIS problem itself, the mapping of the container structure to a discretized model and the description of graph algorithms. In section 3 an approach without the use of a grid is presented. There, the packing problem is solved by using linear programs using a convex container structure with convex obstacles. Finally, section 4 presents some quality and runtime results achieved by these combinatorial approaches.

2 Discrete Approach

2.1 Hardness of the Discrete Trunk Packing Problem

The Trunk packing problem can be discretized in the following way: First, the rotation of the boxes can be limited to axis-oriented placements. Second, the possible placements are restricted to the cells of a grid. Clearly, these restrictions reduce the solution space and possibly eliminate the optimal solution of the original problem. Reichel [11] shows similar problems (DISCRETE-BOX-PACKING for equal boxes) to be $\mathcal{NP}$-complete and the continuous version to be $\mathcal{NP}$-hard. Similar considerations can be applied here.

2.2 Conflict Graph

Reichel presented techniques to compute a grid approximation for the container and uses a graph to describe conflicts between possible box placements on the grid [11]. This can easily be extended to the SAE case. Now there are seven different box types and therefore up to seven nodes for each anchor cell and orientation[2].

SAE J1100 defines seven box types with 17 different side lengths. The finest grid spacing used for the DIN problem was 12.5 mm, which equals about 0.5″. This grid size would be too fine for the SAE case since for each box type, there has to be a copy of each node. Since the conflict graph will be very large for small grid spacings, one has to restrict the minimal spacing to 1″ – 1.5″. Although larger grid spacings lead to smaller conflict graphs, the resulting packings would be larger using small grid spacings such as 1.5″ or 1″.

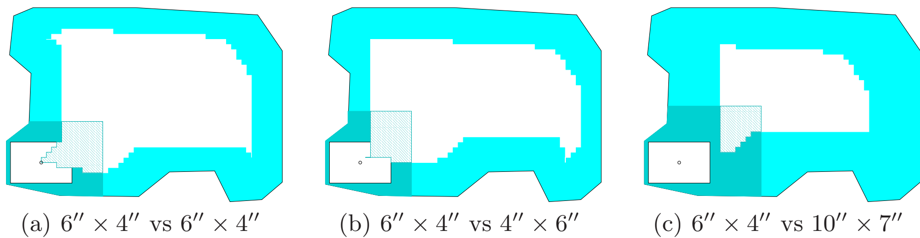
The conflict graph $G = (V, E)$ corresponding to a grid is generated analogue to [5] as follows: For each grid cell (i, j, k), orientation o and box type t, determine whether a box of type t, situated at (i, j, k) in orientation o, would fit into the trunk (i.e. the box would cover only *inside*-cells). If so, then add node $v = (i, j, k, o, t)$ to V. Now, for each pair of nodes $u, v \in V$, check whether the two

[2] Since SAE boxes are much larger than DIN boxes, there are less grid cells useable for a box, and not all box types might fit at a certain position.

corresponding boxes would intersect, i.e. whether there is a cell which is covered by both boxes. If so, then add edge $\{u, v\}$ to E.

Using instances of the DIN 70020 case, the boxes have a size of at most $16 \times 8 \times 4$ cubes (12.5 mm spacing). With 6 possible orientations, each node representing a box might intersect with 201 other possible boxes, thus the conflict graph will have a maximum degree of 201 for 50 mm grid spacing [11]. A typical conflict graph for the DIN problem would have about 10^4 nodes and 10^6 edges (with 50 mm grid spacing). For the same trunk, a conflict graph using $2''$ spacing for the SAE problem has about 8.000 nodes (due to the larger boxes, where less positions are possible) and about $15 \cdot 10^6$ edges (due to the huge number of conflicts for each box). Including the smallest box-type H, the number of nodes in the conflict graph would actually be twice as much. Since the SAE J1100 standard requires the boxes of type H to be packed after the other boxes have been properly packed, it will be sufficient to exclude the H-boxes temporarily. After having computed a solution without H-boxes, the packing can be extended.

The largest box of the SAE problem consists of $12 \times 9 \times 4$ cubes ($2''$ spacing[3]), hence covering 432 cubes. The box intersects with all boxes that overlap one of these cubes which means up to 10^5 potential conflicts in a grid of $2''$ spacing for the biggest box of type A.



(a) $6'' \times 4''$ vs $6'' \times 4''$ (b) $6'' \times 4''$ vs $4'' \times 6''$ (c) $6'' \times 4''$ vs $10'' \times 7''$

Fig. 1. Conflict regions

Figure 1 shows regions of forbidden reference points when one box of size $6'' \times 4''$ is already placed. The forbidden area due to the boundary of the trunk is coloured light grey, and the forbidden area due to the placed box is coloured dark grey. The hatched area marks those points that are allowed by the boundary constraints but placing a box there would cause to intersect the previously placed box.

2.3 Reducing the Conflict Graph Size

Since most of the following algorithms examine the edges of the conflict graph, it might prove useful to *reduce* its size. To find an IS of maximal size resp. maximal

[3] For computational purposes, the SAE box lengths will be rounded down to fit into an integer number of cubes, e.g. the largest box (type A) has $24'' \times 19'' \times 9''$, making a $12 \times 9 \times 4$ grid box. Section 2.8 deals with the problem of rounding the box measures to fit into the grid spacing.

weight, it is necessary to observe some properties of a maximal IS I. The goal is to eliminate some nodes of the graph without reducing the size of the largest IS. In the unweighted case the decision which nodes are to be deleted is quite easy. First, the set of neighbours of a node v has to be defined:

Definition 1. *Let $v \in V$ be a node of G. Then $N(v) = \{u \in V : \{u, v\} \in E\}$ is the set of neighbours of v. The set $N^+(v) = N(v) \cup \{v\}$ denotes the set of neighbours of v including v.*

Now, an elementary property of independent sets can be described.

Theorem 2. *Let $G = (V, E)$ be an undirected graph, and $u, v \in V$ be adjacent nodes ($\{u, v\} \in E$). If $N^+(u) \subsetneq N^+(v)$, then the following is true:*
For every independent set $I \subseteq V$ containing v, there is also an independent set $I' \subseteq V$ with at least the same cardinality containing u.

Proof. Obviously, $v \in I \Leftrightarrow u \notin I$ since $\{u, v\} \in E$. $N^+(u) \subsetneq N^+(v)$ states that all neighbours of u are also neighbours of v, so v can be replaced by u. That is, $I' := I \setminus \{v\} \cup \{u\}$.

Theorem 2 can be used in an elegant way to reduce the nodes of a graph. Since the goal is to find an IS of maximum size, it is useful to include the node u instead of v because v has got a larger neighbourhood and more nodes would be useless for later additions. Since *all* neighbours of u are also neighbours of v, one would rather choose u with its smaller neighbourhood.

What does this mean for the trunk packing problem? If there is a box position near the boundary of the grid, some space between box and boundary would be unusable for other boxes (see figure 1). So it would be better to move the box directly to the boundary, where less space is wasted. This approach is sometimes called a *bottom left justified* packing: All boxes are placed in a way that their bottom, left and front sides touch either other boxes or the surrounding geometry. Any axis-oriented packing can be transformed into a bottom left justified packing [14, 16].

Theorem 2 also applies to the WIS problem:

Corollary 3. *Let $G = (V, E)$ be a weighted undirected graph, and $u, v \in V$ be adjacent nodes ($\{u, v\} \in E$). If $N^+(u) \subsetneq N^+(v)$* and *$w(u) \geq w(v)$, then the following is true:*

For every weighted independent iet $I \subseteq V$ with $v \in I$, there is also a weighted independent set $I' \subseteq V$ with at least the same weight with $u \in I$.

Proof. Simply replace v by u. Then $w(I') = w(I) - w(v) + w(u) \geq w(I)$.

Corollary 3 shows the crucial problem of the SAE case: If two different box types are compared, normally the larger box also gets the larger neighbourhood. So it is not clear which node to eliminate in these cases. Fortunately, there are only seven (a fixed small number) different node weights. So an easy way to reduce the nodes is to do this *typewise*, i.e. to compare only nodes of the same weight. In the general case, one could classify the nodes into a few weight classes and perform the reduction on these classes.

Algorithm 1. Graph reduction
for each type t **do**
  for each edge $\{u, v\}$ with $Type(u) = Type(v) = t$ **do**
    if $N^+(u) \subset N^+(v)$ **then** let $V := V \setminus v$.

Using this procedure it is possible to reduce the number of nodes of a typical conflict graph of the Trunk Packing problem with cell size $2''$ by 75% and the number of edges by up to 90%. Therefore, the performance of the packing algorithms increases significantly because of the smaller conflict graph. The effect of reduceable nodes is directly related to the size of the boxes compared to the trunk. Since the SAE boxes are relatively large and the DIN boxes are relatively small, the reduction algorithm is only useful for the SAE case. When small boxes (type H) are included, only nodes with relatively close positions to each other could be candidates for Theorem 2. Thus, small boxes reduce the positive effects of graph reduction additionally to their large contribution to the number of nodes and edges. The reduction process itself mainly takes place near the boundary of the grid. Since small trunks have got a large boundary compared to their interior, the conflict graphs of smaller trunks can easily lead to graphs of less than 20% of the original graph size. Additionally, when the grid spacing is reduced, the portion of eliminated nodes tends to increase.

2.4 Brute Force Algorithms

The easiest way to find an IS is to successively add a free node to an already existing IS I. A node $v \in V$ is called *free* if there is no edge between v and any node of I. This is done until there are no free nodes left. The SAE case adds the following condition to the definition of free nodes: The number of nodes of the corresponding types must not exceed a pre-defined number, e.g. there may only be four A-boxes and two C-boxes in a valid packing (see Table 1).

This simple approach will hardly lead to the desired optimal solution in the first step. Therefore, a backtracking mechanism has to be added: Test all independent sets of the conflict graph recursively. If there is no free node left, compare the current IS with the best set found so far, and take the better one. Algorithm 2 shows an outline of this procedure. Let therefore $w(v)$ be the weight of node v, and $w(I) = \sum_{v \in I} w(v)$ be the total weight of the set I.

Algorithm 2. Recursive enumeration (G, I)
1. Let F be the set of all free nodes of G
2. **if** $w(I) > w(I_{\max})$ **then** set $I_{\max} := I$.
3. **while** $F \neq \emptyset$ **and** Upper bound $(F, I) > w(I_{\max})$ **do**
   (a) Choose $v \in F$ and set $I' := I \cup \{v\}$, $F := F \setminus \{v\}$
   (b) Recursive enumeration (G, I')
4. **return** $(I_{\max}, w(I_{\max}))$.

Using this approach, the IS of maximum weight will definitely be found since all IS are examined. Unfortunately, the number of independent sets is exponential in the number of nodes. So it is necessary to find an efficient branch-and-bound technique to prune the recursion tree.

2.5 Computing an Upper Bound

Now the main question is: Starting from the current IS, what is the best possible size one could achieve? The exact solution for this problem is equivalent to solving the IS problem on the subgraph induced by all free nodes. Fortunately, the SAE standard gives a hint how to calculate a strong upper bound on the current WIS efficiently: Add the weights of all free nodes but limit the number of the free nodes to the allowed number of boxes for each type. This would mean that the upper bound is calculated at first by using all allowed boxes ($\approx 28.7\,\text{ft}^3$), and will only be reduced if there are less free nodes of a certain type left than allowed.

2.6 A Greedy Algorithm

Up to now, only a brute force algorithm has been presented. Although this approach gives the exact result for the current WIS problem, the runtime still is exponential in $|V|$. Maybe a polynomial-time approximation algorithm can be found which is easy to analyze.

For comparison with the optimal solution of the problem, it is necessary to give an additional definition:

Definition 4. *[13] Let $v \in V$ be a node of G. Then the weighted degree $d_w(v)$ is defined as $d_w(v) = \frac{\sum_{u \in N(v)} w(u)}{w(v)}$.*

The weighted degree of a node shows how much weight would become unuseable in relation to its own weight. Now it is possible to sort all free nodes ascending by their weighted degree within the remaining graph. The greedy algorithm for weighted independent sets works as follows:

Algorithm 3. Greedy (G)
1. Let $I := \emptyset$, $U := V$.
2. **while** $U \neq \emptyset$ **do**
   (a) Let $v \in U$ be a node with $d_w(v) = \min_{u \in U} d_w(u)$.
   (b) $I := I \cup \{v\}$.
   (c) $U := U \setminus N(v)$.
   (d) Recompute $d_w(u)$ for all $u \in U$ w.r.t. U (not V).

In [13] an approximation ratio of $\min((\bar{d}_w + 1)/2, (\delta_w + 1)/2)$ for the greedy algorithm is derived. It is obvious that an asymptotic bound is not really useful for the small cases of the Trunk Packing problem. It also comes clear that the greedy algorithm achieves more than the expected asymptotic bound for graphs derived from trunk packing instances.

2.7 Greedy Enumeration

Observing these facts it will be necessary to extend algorithm 3 by algorithm 2 to enumerate all weighted IS systematically.

Greedy! Line 2d of algorithm 3 needs an efficient update function, especially if it is planned to use this approach for a recursive enumeration of all weighted independent sets. If node v is added to the IS, then all nodes of $N(v)$ would be unuseable. Hence, the weighted degrees of all nodes $w \in N(N(v))$ will be decremented by $w(N(w) \cap N(v))$. In the present implementation, all pairs u, w of nodes of the remaining graph are examined, where $u \in N(v), w \notin N(v)$ and $\{u, w\} \in E$.

Node orders. For an enumeration algorithm, the node order is crucial. If the right sequence is chosen, a very good result can be achieved at an early stage, and in combination with a good upper bound, the enumeration process can be cut early. In the previous sections, no specific order of the vertices is assumed. Normally, they are sorted by their time of creation in the conflict graph. In the current implementation the nodes are first sorted by type, then by orientation and position. For instance, a random permutation might be chosen and the recursive algorithm executed like a multi-start approach, increasing the probability to find the optimal solution earlier. Again, this approach is difficult to analyze. The experiments showed a large average runtime for the randomized node order. Another possibility would be taking nodes with small weighted degree first. In the trunk packing problem, the weighted degrees are directly correlated to the sizes of the boxes, hence a node of small weighted degree represents a small box. Therefore, this strategy means packing small boxes first.

For the SAE Trunk Packing problem the following observation can be made: If a large box is taken first, it will cause a large amount of nodes to be unuseable and therefore the remaining conflict graph will be significantly smaller. Also, a large box has only few possibilities to be placed, making the recursion tree narrower. But if a small box is chosen, the number of possibilities will be much larger and the algorithm will be stuck at an early stage by examining many almost equal sets. This approach is very alike a human's way to pack a trunk: take the big cases first, and then stuff the smaller boxes in between. Now an additional effect comes into account: If the nodes with large weights are used first and the small ones afterwards, it is possible that the last few (small) nodes can not exceed the best found IS, and the algorithm can make effective use of the upper bound discussed in section 2.5. This implementation proved to be the fastest in all experiments.

2.8 Rounding the Box Sizes

Another discouraging aspect is the need for rounding the box sizes. In a $2''$-grid, only 8 of 21 side lengths are represented correctly (only the even ones), and all other lengths would have to be rounded. Table 1 shows that the largest grid spacing without rounding the box sizes would be $0.1''$, thus far from practicable. Now there are some different possibilities to be discussed:

1. All lengths are rounded up to the next even integer.
2. All lengths are rounded down to the last even integer.
3. All lengths are rounded to the nearest even integer.
4. The lengths are rounded up or down with respect to the available space.

The first strategy will provide a *feasible*[4] solution when the set is converted into a real box set. Unfortunately, this strategy will waste space between the boxes. Strategy 2 causes the opposite problem. Since the grid boxes are smaller than the real boxes the resulting solution will be unfeasible due to intersections between boxes. These intersections can be resolved by using a physical simulation of contacts between the boxes and the trunk [1]. However, the intersections might be too severe to be resolved. To prevent this, the outmost layer of *inside*-cells is deleted. This provides a feasible solution in most cases. Strategy 3 will cause both problems of the first two rounding strategies. A mixed strategy would raise the problem of data representation. Currently, strategy 2 is used. In most cases it suffices to delete one layer of *inside*-cells in each dimension to generate legalizable solutions, i.e. the resulting packings themselves would not fit into the trunk, but the packed boxes could be translated and rotated to fit into the trunk. However, this reduction of the grid causes another deficit in comparison to a manual solution.

3 Omitting the Grid

An optimal solution with respect to a grid is limited to fixed orientations and discretized placements. Thus, such a solution is unlikely to be optimal with respect to the original trunk packing problem. If the restriction to discrete placements is dropped, it might be possible to find a better solution. Omitting the grid means that the box sizes are not required to be multiples of a grid cell size and therefore do not have to be rounded. So a legalization step is not necessarily required.

Schepers [14] shows how to compute packings of axis-oriented boxes within a cubic container of "unit" length. This is done by solving several linear programs.

For a fixed set of boxes we use enumeration to decide the relative position of the boxes – i.e. for each pair of boxes, we decide whether the first box lies left, right, above (but neither left nor right), below, in front of, or back of the other box. Using the center coordinates of each box, the relative positions between boxes can easily be enforced by linear inequalities. Furthermore we can describe the set of all feasible points for the centers of the boxes by linear inequalities.

Hence the feasibility of such a configuration, called packing pattern, can be tested by verifying the feasibility of a linear program. Schepers additionally shows how to solve these linear programs combinatorially, but this result does not generalize to our approach. The optimal set of boxes is found by enumerating all sets of boxes (such that a subset fits into the container).

Let $T \subset \mathbb{R}^3$ be the interior of the trunk. For a box B with fixed type and orientation, let T_B be the Minkowski sum of the trunk boundary and B. Now the set $M = T \setminus T_B$ describes all reference points where box B can be anchored within the trunk.

To generalize the approach described above to the case of a trunk, the set M has again to be described by linear inequalities. As this set is not convex, we

[4] A packing is considered feasible if there are no intersections between the packed boxes as well as between the boxes and the trunk.

over-approximate the set M by a convex region and allow to place some convex bodies, called obstacles, into the set. Similar to the case of two boxes, we have to know the relative position of every box and every obstacle, i.e. knowing which of the half spaces defining the convex obstacle does not intersect the center of the box.

Notice that the complexity of the approach grows rapidly with the number of half spaces describing the inserted obstacles. Therefore we aim to describe a close approximation of the feasible region for the centers by a small number of obstacles.

Currently this description is done manually. First, the trunk itself is approximated by linear equations, and then cuboid obstacles are inserted. It is quite easy to calculate the Minkowski sum of a set of linear equations and a box. Especially the desired set $M = T \setminus T_B$ can be described as the same linear equations translated by the half diagonal of the box. Since the obstacles are restricted to be convex, they can be described in the same way. At the moment, we develop automated methods to describe the set M in the required way.

To improve the practical efficiency of this approach, we enumerate the region, the inequalities of an obstacle and the relative position only if the appropriate constraints are violated by our solution of the linear program.

As the number of feasible packing patterns and therefore the number of linear programs solved increases rapidly with the size of the trunk, the enumeration of all packing patterns is very time consuming. For the manually created approximations, we are able to enumerate all packing patterns within a few days. Moreover, we experimented with the following heuristic. We first enumerate packings that use only large boxes. Then we take some of the best packings and add smaller boxes if possible.

4 Experimental Results

All grid based algorithms discussed so far are exact, so they will find the optimal solution for the given discretization of the trunk. Table 2 shows the best achieved results of some typical instances. As can be seen, the results achieved by the LP approach are comparable to those provided by the grids. Very fine grids cause a too large conflict graph for an efficient computation. Also, the finer grids do not necessarily provide the best grid-based solutions, contrary to the DIN case [5]. This follows from the rounding strategy described in section 2.8. Rounding the box sizes according to a large grid spacing leads to larger intersections between the packed boxes. Anyway, it is possible in most cases to resolve these intersections using a physical contact simulation [1]. The best grid-based results compared to the computational effort could be achieved using the $2''$ and $1.5''$ spacings.

The pre-computation of the node order proved quite useful. If the nodes are sorted by their weighted degrees only, the problem of calculating the upper bound remains: One has to scan all free nodes whether there still are nodes of a certain type left or not. If the nodes are sorted by their type, this examination

Table 2. Experimental results for some problem instances without H-boxes

trunk	manually	grid-based (algorithm 2)				LP-based
		$3''$	$2''$	$1.5''$	$1''$	
small	5.337 ft^3	4.767 ft^3	5.071 ft^3	5.648 ft^3	5.392 ft^3	5.792 ft^3
mid-size	9.358 ft^3	8.538 ft^3	9.207 ft^3	9.418 ft^3	9.041 ft^3	9.521 ft^3
large	11.837 ft^3	10.871 ft^3	12.056 ft^3	12.674 ft^3	11.790 ft^3	12.637 ft^3

Table 3. Runtimes needed by various grid based strategies for exact solution (without H-boxes)

trunk	grid	nodes/edges	timestamp	randomized	sorted	greedy
small	$1.5''$	$470/7.8 \cdot 10^4$	$5sec$	$2.5sec$	$< 1sec$	$< 1sec$
small	$1''$	$850/2.5 \cdot 10^5$	$1m40sec$	$20sec$	$< 1sec$	$2sec$
mid-size	$3''$	$480/6.3 \cdot 10^4$	$45sec$	$3m$	$1.75sec$	$7sec$
mid-size	$2''$	$1200/4 \cdot 10^5$	$40m$	$1h40m$	$10sec$	$1m$
mid-size	$1.5''$	$2200/1.4 \cdot 10^6$	$> 24h$	$> 24h$	$3m20sec$	$13m$
large	$3''$	$660/1.1 \cdot 10^5$	$6m$	$10h$	$6sec$	$30sec$
large	$2''$	$2000/9.6 \cdot 10^5$	$18h$	$> 24h$	$17sec$	$1m30sec$
large	$1.5''$	$4800/5.9 \cdot 10^6$	$> 24h$	$> 24h$	$8m20sec$	$45m$

is obsolete: If the algorithm adds a node of type t, then it is guaranteed that no more nodes of previous types are free. This leads to an additional time saving effect. The runtimes needed for verifying the optimal solution are compared in Table 3. Within this context, verifying an optimal solution implies a complete run of the algorithm.

The algorithms `timestamp`, `randomized` and `sorted` refer to algorithm 2 and use different node orders: the time of creation, a randomized order and sorted by box size and weighted degree, respectively. Algorithm `greedy` refers to the recursive greedy algorithm discussed in section 2.7. All grid algorithms were terminated after 24 hrs due to the given time constraints. It is easy to see that the order of the nodes plays an important role in the search for the best possible IS.

The LP algorithm provided exact solutions for a given approximation only for small instances. Otherwise, the algorithm was stopped after 24 hours runtime. Table 2 shows that both approaches, the grid-based and the LP-based approach, are almost equal with slight quality advantage on the LP side. On the other hand, the grid-based algorithms are easier to operate.

5 Summary

In this paper the trunk packing problem for the US standard SAE J1100 has been investigated. Two combinatorial approaches were presented: First, a discretization of the space to be packed, analogous to [5, 11] for the DIN case. Second, an approximation scheme using linear inequalities. For both approaches

enumerative algorithms have been described. As shown in section 4, the brute-force algorithms suffice for the trunks used in car design processes to enumerate all possible solutions for the resulting Weighted Independent Set problem. This could be reached by reducing the conflict graph and applying a property provided by independent sets.

Unfortunately, a discretization of the space leads to insufficient representation of the SAE-boxes. So additional methods have to be used to overcome illegal situations such as box-box intersections. These methods include a contact simulation provided by a physics engine [1].

The LP approach needs a space consisting of few convex regions and only few convex obstacles. So far no automated scheme is known to provide such simplifications for a complicated geometric structure like a trunk. Additionally, the enumerative algorithms are exponential in runtime. This means that large instances (e.g. small boxes into a large irregular shaped container) can only be solved approximately.

Further work includes more efficient graph reduction algorithms or a fast exact algorithm for the WIS problem capable of handling large trunk instances and fine grids. For the LP algorithm it is necessary to find a good approximation of the trunk to ensure the resulting packing to fit into. Also an improved heuristic for large instances is needed.

References

[1] Baumann, T., Schömer, E., Werth, K.: Solving geometric packing problems based on physics simulation. Submitted for ESA'07 (2007)

[2] Berman, P.: A d/2 approximation for maximum weight independent set in d-claw free graphs. Nordic J. of Computing 7(3), 178–184 (2000)

[3] Cagan, J., Ding, Q.: Automated trunk packing with extended pattern search. Virtual Engineering, Simulation & Optimization SP-1779, 33–41 (2003)

[4] Eisenbrand, F., Funke, S., Karrenbauer, A., Reichel, J., Schömer, E.: Packing a trunk: now with a twist! In: SPM '05: Proceedings of the 2005 ACM symposium on Solid and physical modeling, pp. 197–206. ACM Press, New York (2005)

[5] Eisenbrand, F., Funke, S., Reichel, J., Schömer, E.: Packing a trunk. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 618–629. Springer, Heidelberg (2003)

[6] Deutsches Institut für Normung e.V. DIN 70020, Teil 1, Straßenfahrzeuge; Kraftfahrzeugbau; Begriffe von Abmessungen (February 1993)

[7] Kako, A., Ono, T., Hirata, T., Halldórsson, M.M.: Approximation algorithms for the weighted independent set problem. In: Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG, pp. 341–350 (2005)

[8] Karrenbauer, A.: Packing boxes with arbitrary rotations. Master's thesis, Universität des Saarlandes, Saarbrücken (2004)

[9] Neumann, U.: Optimierungsverfahren zur normgerechten Volumenbestimmung von Kofferräumen im europäischen Automobilbau. Master's thesis, Technische Universität Braunschweig, Braunschweig (2006)

[10] Society of Automotive Engineers. SAE J1100, Motor Vehicle Dimensions (February 2001)

[11] Reichel, J.: Combinatorial approaches for the Trunk packing problem. PhD thesis, Saarbrücken (2006)
[12] Rieskamp, J.: Automation and Optimization of Monte Carlo Based Trunk Packing. Master's thesis, Universität des Saarlandes, Saarbrücken (2005)
[13] Sakai, S., Togasaki, M., Yamazaki, K.: A note on greedy algorithms for the maximum weighted independent set problem. Discrete Appl. Math. 126(2-3), 313–322 (2003)
[14] Schepers, J.: Exakte Algorithmen für orthogonale Packungsprobleme. PhD thesis, Köln (1997)
[15] Warren, J.S., Hicks, I.V.: Combinatorial branch-and-bound for the maximum weight independent set problem. http://ie.tamu.edu/people/faculty/Hicks/jeff.rev.pdf (2006)
[16] Wottawa, M.: Struktur und algorithmische Behandlung von praxisorientierten dreidimensionalen Packungsproblemen. PhD thesis, Köln (1996)
[17] Zimmermann, A.: AI Zimmerman's Circle Packing Contest. Al Zimmerman's Programming Contests, http://www.recmath.org/contest/CirclePacking/ (2005)

Exact Algorithms for the Matrix Bid Auction*

D.R. Goossens and F.C.R. Spieksma

Dept. of Operations Research, Katholieke Universiteit Leuven, Belgium

Abstract. In a combinatorial auction, multiple items are for sale simultaneously to a set of buyers. These buyers are allowed to place bids on subsets of the available items. A special kind of combinatorial auction is the so-called matrix bid auction, which was developed by Day (2004). The matrix bid auction imposes restrictions on what a bidder can bid for a subsets of the items. This paper focusses on the winner determination problem, i.e. deciding which bidders should get what items. The winner determination problem of a general combinatorial auction is NP-hard and inapproximable. We discuss the computational complexity of the winner determination problem for a special case of the matrix bid auction. We compare two mathematical programming formulations for the general matrix bid auction winner determination problem. Based on one of these formulations, we develop two branch-and-price algorithms to solve the winner determination problem. Finally, we present computational results for these algorithms and compare them with results from a branch-and-cut approach based on Day and Raghavan (2006).

1 Introduction

In an auction where multiple bidders are interested in multiple items, it is often the case that the value of a set of items is higher or lower than the sum of the values of the individual items. These so-called complementarity or substitution-effects, respectively, may be bidder-specific. A combinatorial auction is a way to make use of this phenomenon. In such an auction a bidder is allowed to place bids on a subset of the items, sometimes called a *bundle*. The auction is concluded when the auctioneer decides to accept some of the bids and to allocate the items accordingly to the bidders.

In a combinatorial auction in its most general form, bidders can bid whatever amount they please on any subset of items. The problem of deciding which bidders should get what items in order to maximize the auctioneer's revenue is called the winner determination problem. This problem is NP-hard (see e.g. Sandholm (2002)), and remains so even if every item occurs in at most two bids and all prices are equal to 1 (see Van Hoesel and Müller (2001)).

The matrix bid auction, developed by Day (2004), is a combinatorial auction in which restrictions are imposed on what a bidder can bid. In this auction, each bidder must submit a strict ordering (or ranking) of the items in which he[1] is

* This research was partially supported by FWO Grant No. G.0114.03.

[1] He can be replaced by she (and his by her).

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 433–445, 2007.
© Springer-Verlag Berlin Heidelberg 2007

interested. We assume that for each bidder, the extra value an item adds to a set is determined only by the number of higher ranked items in that set, according to the ranking of that bidder.

Let G be the set of items, indexed by i and B the set of bidders, indexed by j. The ordering of the items is denoted by r_{ij}, which is item i's position in bidder j's ranking, for each $i \in G$ and $j \in B$. This ordering should be strict in the sense that for each bidder j, $r_{i_1 j} \neq r_{i_2 j}$ for any pair of distinct items i_1 and i_2. For instance, if $r_{ij} = 2$, item i is bidder j's second highest ranked item. Furthermore, each bidder j specifies values b_{ijk}, which correspond to the value the bidder is prepared to pay for item i given that it is the k-th highest ranked item in the set that bidder j is awarded. The b_{ijk} values allow to determine the value bidder j attributes to any set $S \subseteq G$. Indeed, bidder j's bid on a set S is denoted as $b_j(S)$ and can be computed as:

$$b_j(S) = \sum_{i \in S} b_{i,j,k(i,j,S)} \tag{1}$$

where $k(i, j, S)$ is the ranking of item i amongst the items in the set S, according to bidder j's ranking. Notice that equation (1) assumes that no externalities are involved, i.e. a bidder's valuation depends only on the items he wins, and not for instance on the identity of the bidders to whom the other items are allocated. The winner determination problem is, given the bids $b_j(S)$ for each set S and each bidder j, to determine which bidder is to receive which items, such that the total winning bid value is maximized. Notice that we assume that each bidder pays what he bids for the subsets he wins.

Observe that the value for index k of item i in bidder j's bid can never be higher than the rank r_{ij}. This allows us to arrange the values b_{ijk} as a lower triangular matrix for each bidder j, where the rows correspond to the items, ordered by decreasing rank and the columns correspond to values for k. Hence the name matrix bid (with order). Notice also that bidder j's ranking r_{ij} does not necessarily reflect a preference order of the items. If an item is highly ranked, this merely means that its added value to a set depends on less items than the added value of a lower ranked item. Furthermore, we make no assumption regarding the b_{ijk} values. Indeed, these values may be negative, e.g. to reflect the disposal cost of an unwanted item. Specifying a sufficiently large negative value can also keep the bidder from winning this item in the first place. For a more elaborate discussion of the expressiveness of matrix bids and their relation to well-known micro-economic properties, we refer to Goossens (2006).

As a frivolous example, we consider the following matrix bid, where a bidder expresses his preferences for an ice cream. There are two flavors of ice cream (vanilla and banana), and also hot chocolate and strawberry sauce are available.

vanilla ice	4			
banana ice	5	2		
hot chocolate	-5	0	3	
strawberry sauce	-5	0	3	-1

Consider now the value this bidder j attributes to vanilla ice with hot chocolate. Observe that for this choice of S, vanilla ice is the highest ranked item (that is, $k(\textit{vanilla ice}, j, S) = 1$), and hot chocolate is the second highest ranked item (that is, $k(\textit{hot chocolate}, j, S) = 2$). We find using (1):

$$\begin{aligned} b_j(S) &= b_{vanilla\ ice,j,k(vanilla\,ice,j,S)} + b_{hot\ chocolate,j,k(hot\,chocolate,j,S)} \\ &= b_{1,j,1} + b_{3,j,2} \\ &= 4 + 0 = 4. \end{aligned}$$

Thus, this matrix bid can be interpreted as follows: bidder j feels that he needs at least one scoop of ice cream of one of the two available flavors, although he prefers banana. Indeed, no combination without ice cream will result in a positive valuation, because the bidder charges a (disposal) cost of 5 if he gets one or both toppings without ice cream. Furthermore, the bidder is not willing to pay as much for the second scoop of ice cream as for the first. The highest bid this bidder places is 9, for the combination of vanilla and banana ice with any one of the two toppings.

1.1 Motivation

There are several reasons for investigating a combinatorial auction with matrix bids. First, bids in any practical combinatorial auction are likely to posses some structure. In literature, we find descriptions of both theoretical structures (see e.g. Rothkopf et al. (1998), Nisan (2000), and Leyton-Brown and Shoham (2005)), and structures in practice (see e.g. Bleischwitz and Kliewer (2005) and Goossens et al. (2007)). Capturing and understanding this structure is important, not only since it allows to develop algorithms that can be more efficient than algorithms for a general combinatorial auction, but also because it improves our understanding of various properties of an auction.

Second, matrix bid auctions allow for a faster computation due to the restriction on the preferences that is assumed. Indeed, from their computational experiments, Day and Raghavan (2006) conclude that the computation time for the general combinatorial auction is higher and grows much faster than for the matrix bid auction. Moreover, they manage to solve the winner determination problem for matrix bid auctions with 72 items, 75 bidders and over 10^{23} bids, whereas for the general combinatorial auction, the largest instances that can be solved have 16 items, 25 bidders, and less than 10^9 bids.

Finally, the matrix bid auction also offers a compact way of representing preferences. Indeed, each bidder only needs to communicate an ordered list of m items and $\frac{m(m+1)}{2}$ matrix bid entries, which is far less than bids for each of the 2^m possible sets of items in a general combinatorial auction. We do recognize that choosing a ranking of the items and filling the matrix bid with appropriate values might not be a trivial task for the bidder. However, we developed a procedure that recognizes whether a given collection of bids can be translated into a matrix bid, and – in case this is not possible – an algorithm that approximates this collection of bids by a matrix bid in a way that does not expose the bidder to paying more than he stated for any set of items (see Goossens (2006)).

1.2 Our Contribution

In section 2, we strengthen a result by Day (2004) that shows that the matrix bid auction winner determination problem is NP-hard, by studying a special case where all bidders have an identical ranking of the items. We show that this problem is APX-hard. However, given a fixed number of bidders, it can be solved in polynomial time. We discuss a mathematical programming formulation by Day (2004), and compare it to a formulation based on the set packing problem. We find that both formulations are equally strong. Based on the set packing formulation, we develop two branch-and-price algorithms to solve the winner determination problem in section 3. Finally, in section 4, we present our computational results and compare them with results from the branch-and-cut approach by Day (2004).

2 Complexity and Formulations

The key assumption in the matrix bid auction is that for each bidder, the extra value an item adds to a set depends only on the number of higher ranked items in that set, according to the ranking of that bidder. Despite this restriction, the winner determination problem of the matrix bid auction remains NP-hard Day (2004). We strengthen this result by showing that the problem is APX-hard, even under identical rankings.

Theorem 1. *There exists no polynomial-time approximation scheme for the winner determination problem for the matrix bid auction even when the ranking of the items is identical for each bidder, unless* $P = NP$.

In Theorem 1, the number of bidders is part of the input. In the case that the number of bidders is fixed (and we still assume identical rankings), the winner determination problem can be solved in polynomial time, as witnessed by the following result.

Theorem 2. *The winner determination problem for a matrix bid auction with a fixed number of bidders, all having an identical ordering of the items, can be solved by solving a shortest path problem.*

Let us now present two mathematical formulations for the matrix bid auction winner determination problem.

We define the binary variable x_{ijk} to be 1 if bidder j receives item i as the k-th best item, and 0 otherwise. This leads to the formulation below, to which we refer as the assignment formulation and which was originally developed by Day (2004).

$$\text{maximize} \sum_{i \in G} \sum_{j \in B} \sum_{k=1}^{r_{ij}} b_{ijk} x_{ijk} \tag{2}$$

$$\text{subject to} \sum_{j \in B} \sum_{k=1}^{r_{ij}} x_{ijk} \leqslant 1 \qquad \forall i \in G \tag{3}$$

$$\sum_{i \in G: r_{ij} \geq k} x_{ijk} \leqslant 1 \qquad \forall j \in B, \forall k \tag{4}$$

$$\sum_{l \in G: k \leq r_{lj} \leq r_{ij}} x_{ljk} \leqslant \sum_{l \in G: k-1 \leq r_{lj} < r_{ij}} x_{ljk-1} \quad \forall i \in G, \forall j \in B, \forall k \geq 2 \tag{5}$$

$$x_{ijk} \in \{0,1\} \qquad \forall i \in G, \forall j \in B, \forall k \tag{6}$$

Constraints (3) enforce that each item can be assigned to at most one bidder, while constraints (4) make sure that for each bidder, at most one item is the k-th best item in the set this bidder gets. Finally, constraints (5) impose that a bidder cannot get an item as the k-th best item in a set, unless a higher ranked item was assigned to this bidder as his $(k-1)$-th best item in this set. Constraints (6) are the integrality constraints.

Notice that the formulation (2)-(6) is not the minimal correct formulation for the matrix bid winner determination problem. Indeed, constraints (4) for $k \in \{2, ..., r_{ij}\}$ are redundant in (2)-(6), since they are already enforced by constraints (4) for $k = 1$ and constraints (5). Also, replacing constraints (5) with the following (weaker) constraints still results in a correct formulation:

$$x_{ijk} \leqslant \sum_{l \in G: k-1 \leq r_{lj} < r_{ij}} x_{ljk-1} \qquad \forall i \in G, \forall j \in B, \forall k \geq 2.$$

However, with this formulation, all constraints (4) remain necessary.

The set packing formulation below makes use of binary variables $y(S, j)$, which equals 1 if bidder j wins set S, and 0 otherwise. The first set of constraints (8) enforces that each item is awarded to at most one bidder. The second set of constraints guarantees (9) that no bidder receives more than one set. The integrality constraints are (10).

$$\text{maximize} \sum_{j \in B} \sum_{S \subseteq G} b_j(S) y(S, j) \tag{7}$$

$$\text{subject to} \sum_{S \supseteq \{i\}} \sum_{j \in B} y(S, j) \leqslant 1 \qquad \forall i \in G \tag{8}$$

$$\sum_{S \subseteq G} y(S, j) \leqslant 1 \qquad \forall j \in B \tag{9}$$

$$y(S, j) \in \{0, 1\} \qquad \forall S \subseteq G, \forall j \in B \tag{10}$$

Notice that this set packing formulation can also be used for the winner determination problem of a general combinatorial auction. Indeed, the matrix bid

auction only differs from a general combinatorial auction in the way $b_j(S)$ is computed. Notice also that the assignment formulation is polynomially sized in the number of bidders and the number of items. This is not the case for the set packing formulation. In the following theorem, we prove that the LP-relaxation of the set packing formulation and the LP-relaxation of the assignment formulation are equally strong.

Theorem 3. *The LP relaxation of the assignment formulation and the LP relaxation of the set packing formulation are equally strong. Moreover, if the assignment formulation has an integral solution that is optimal with respect to the LP-relaxation, this is also the case for the assignment formulation, and vice versa.*

3 Branch-and-Price Algorithms for Solving the Matrix Bid Auction

Here we outline an algorithm based on the set packing formulation. Solving the LP-relaxation of the set packing formulation is however not trivial, given the huge amount of variables ($n2^m$). Considering that only a small percentage of these variables are nonzero in an optimal solution, column generation suggests itself as an efficient solution technique. Notice that this problem can be restricted to $m + n$ variables, whereas the assignment formulation requires $nm(m + 1)/2$ variables, which may still be large. Next, we solve the so-called pricing problem iteratively until the LP-relaxation has been solved to optimality.

3.1 Column Generation for the Matrix Bid Auction

In this section, we show how the LP-relaxation of the set packing formulation of the matrix bid winner determination problem can be solved using column generation. We also prove that the pricing problem can be solved in polynomial time, since it can be reduced to a shortest path problem.

If we define u_i for each item $i \in G$ as the dual price associated with the corresponding constraint of (8), and v_j for each bidder $j \in B$ as the dual price associated with the corresponding constraint of (9), it follows that the pricing problem boils down to determining the existence of a set S of items and a bidder j such that

$$\sum_{i \in S} u_i < b_j(S) - v_j. \tag{11}$$

Theorem 4. *The pricing problem can be solved by solving a shortest path problem.*

Thus, if the shortest path has a negative length, we can add a column for the corresponding bidder j containing the items in set S determined by the item nodes traversed in the path. Naturally, bidder j's bid for this set S is $b_j(S)$. Notice

that since the pricing problem is solvable in polynomial time, the LP-relaxation of the set packing formulation for the matrix bid auction can also be solved in polynomial time. Indeed, this follows from the "separation = optimization" result by Grötschel et al. (1988).

3.2 Branching on an Item-Bidder Pair

The solution of the LP-relaxation of the matrix bid winner determination problem may not be integral. If this is the case, we need to partition the solution space to eliminate this fractional solution. In this approach, we partition the solution space by the branching decision whether or not to assign an item to a bidder. We first prove that in a fractional solution, there always exists an item that has been fractionally assigned to one or more bidders.

Lemma 1. *For any fractional solution to the relaxation of (7)-(10),*

$$\exists i \in G, j \in B : 0 < \sum_{S:S \supseteq \{i\}} y(S,j) < 1 \tag{12}$$

We refer to Goossens (2006) for a description of the branching rule based on this lemma. Using this rule, the pricing problem remains solvable as a shortest path problem throughout the search tree.

3.3 Branching on a Pair of Successive Items

In the spirit of Ryan and Foster (1981), another branching rule is possible. This rule is based on the following property of an extreme optimal LP-solution.

Lemma 2. *For any optimal, extreme fractional solution to the relaxation of (7)-(10),*

$$\exists p, q \in G : 0 < \sum_{j \in B} \sum_{S:S \supseteq \{p,q\} \wedge p \rightarrow_j q} y(S,j) < 1 \tag{13}$$

Again, we refer to Goossens (2006) for a description of the branching rule based on this lemma. And again, using this rule, the pricing problem remains solvable as a shortest path problem throughout the search tree.

3.4 Implementation Issues

Both branch-and-price algorithms were implemented using Visual C++ 6.0. The set packing problems were solved using Ilog Cplex 8.1. The LEDA libraries (version 5.0.1) allowed us to solve the shortest path problems in linear time. In the remainder of this section, some of the most important implementation issues are discussed.

Solving the Root Node. A first issue that needs to be solved is determining which columns will be used in the very first restricted master problem. Using many columns obviously increases the computation time needed to solve the restricted master problem. On the other hand, this may result in a solution that

is closer to the optimal solution, such that less iterations for solving the pricing problem and re-optimizing are needed. In our case, after experimenting with a number of settings, it turned out that including a rather large number of variables to start the column generation process pays off. We constructed a set for every strictly positive entry in the matrix bid by taking the item corresponding to this entry and completing the set with the k highest ranked items, where k is the entry's column in the matrix bid.

After the restricted master problem has been solved and the corresponding dual solution has been obtained, new columns with a non-negative reduced cost need to be added. The question remains how many such columns we should add. Again, adding too many new variables increases the computation time for solving the resulting restricted master problem, whereas adding too few variables can result in a large number of iterations for solving the pricing problem and re-optimizing. The strategy that proved to be the most efficient consists of adding for each bidder those variables whose reduced cost is at most 2% less than the most positive reduced cost for a variable from that bidder. Furthermore, the number of such variables that is added for each bidder cannot exceed the number of items. Notice that finding these variables demands very little extra computation time, since the LEDA libraries provide the distance from the source to each node in the graph, after having solved the shortest path problem.

Finally, when re-optimizing the restricted master problem, we start from the optimal base of the previous iteration. In order not to drag along too many columns for the remainder of the search tree, those columns that were added at some iteration, but never made part of any base solution are removed from the model. We keep the other columns, assuming that they will be useful again later.

A Selection Rule When Branching on an Item-Bidder Pair. The major issue in implementing this branching rule is to choose the item on which to branch and the bidder(s) to assign it to. We chose to branch on the item that is fractionally assigned to the highest number of bidders. For each of these bidders, a branch is constructed in which the bidder is assigned the item. A final branch is added where none of these bidders is allowed to receive the item. We opted for a depth-first strategy, where the branch where the item is assigned to the bidder with the highest fraction is explored first. Thus, the branch where bidders are disallowed to receive an item always comes last.

A Selection Rule When Branching on a Pair of Successive Items. With this branching rule, each node that needs further partitioning of the solution space leads to two branches. In the first branch, we enforce that for each bidder, if item p is present in a bid, q should be the next item in that bid, according to the ranking of that bidder. The second branch considers only bids for which p and q are no direct successors according to the bidder's ranking. We again chose a depth-first strategy, where the branch where $p \rightarrow_j q$ is imposed is explored first. The question remains how to select the items p and q. We opted to pick those items p and q for which $\sum_{j \in B} \sum_{S:S \supseteq \{p,q\} \wedge p \rightarrow_j q} y(S,j)$ is closest to 0.5.

Solving a Tree Node. Before we can start solving a node of the tree, we remove all columns that do not satisfy the latest branching decision. In case of backtracking, this branching decision expires and those columns are re-entered into the model, since we experienced that they often turn out to be useful in other branches of the tree.

The LP objective value of the node can be used as an upper bound to the integral solution that could be found further down the tree. Clearly, if this value is lower than the incumbent found so far, the node can be pruned. We also use the Lagrangian upper bound:

$$\delta + \sum_{j \in B} \max_{S \subseteq G}(RC(S,j),0), \tag{14}$$

where δ is the objective value of the restricted master and $RC(S,j)$ is the reduced cost of variable $y(S,j)$ (Vanderbeck and Wolsey (1996)). If at any iteration in the column generation process, the Lagrangian upper bound is lower than the incumbent, we can prune the node, without any risk of missing the optimal solution.

4 Computational Results

In this section, we elaborate on how we generated the instances on which the branch-and-price algorithms were tested. We also give an overview of the computational results and compare them with results from a branch-and-cut approach performed on the assignment formulation.

4.1 Structure of the Instances

Unfortunately, real-life data for combinatorial auctions are not abundantly available for the public. It is therefore not uncommon to turn to randomly generated data (see for instance Leyton-Brown et al. (2000), Sandholm (2002), and Parkes (1999)). For a thorough discussion on the empirical hardness of several data distributions commonly used for combinatorial auctions, we refer to Leyton-Brown et al. (2005).

For our instances, each matrix bid is composed according to a bid type, randomly chosen out of the six possibilities discussed in Day (2004) (additive preference bids, single-minded bids, nested flat bids, nested k-of bids, partition bids, and add-on bids) and a bid type that has non-increasing rows and columns. In order to avoid auctions for which the exact solution of the winner determination problem is obvious, the matrix bids are constructed such that they are competitive. Furthermore, there is a parameter H that bounds the highest incremental value an item brings to a set. For more details on the bid types or on how the instances were generated, we refer to Day (2004).

We performed experiments on matrix bid auctions with 5, 10, 25 or 50 items and 5, 10, 25, 50, 75 or 100 bidders. For each combination, 10 instances were generated and solved to optimality. The highest incremental value per item (H)

was limited to 10. We have no indication that the branch-and-price algorithm performs differently with other settings for H. All computational experiments were done on a desktop computer with a Pentium IV 2 GHz processor, with 512 MB RAM.

4.2 Results

We report both average and median computation times of three approaches. Indeed, it is not uncommon in literature on combinatorial auctions to study the median (see for instance Sandholm et al. (2005) and Hoos and Boutilier (2000)).

Table 1. Average computation times [s] for n bidders and m items using BOI

	$n = 5$	10	25	50	75	100
$m = 5$	0.005	0.007	0.008	0.017	0.027	0.038
10	0.027	0.038	0.053	0.088	0.118	0.169
25	0.636	0.597	1.157	4.292	12.704	49.155
50	247.224	60.711	437.951	557.083	622.591	802.483

Table 2. Average computation times [s] for n bidders and m items using BOS

	$n = 5$	10	25	50	75	100
$m = 5$	0.005	0.006	0.006	0.018	0.027	0.038
10	0.033	0.037	0.044	0.067	0.104	0.182
25	0.698	0.767	1.194	3.814	16.300	97.122
50	76.598	67.584	843.435	259.079	645.632	983.539

Table 3. Average computation times [s] for n bidders and m items using B&C

	$n = 5$	10	25	50	75	100
$m = 5$	0.030	0.027	0.049	0.052	0.070	0.102
10	0.050	0.069	0.140	0.278	0.524	0.748
25	0.757	1.391	3.598	10.689	17.584	31.940
50	57.676	28.333	91.230	215.083	355.785	811.960

Tables 1 and 2 give an overview of the average computation times needed to solve the matrix bid auction winner determination problem using branch-and-price with branching on an item-bidder pair (BOI) and branch-and-price with branching on a pair of successive of items (BOS) respectively. In Table 3, we give the average computation times that resulted from solving the assignment based formulation (2)-(6) with the Ilog Cplex 8.1 branch-and-cut algorithm with standard settings (B&C), which is basically the approach followed in Day and Raghavan (2006). Tables 4 to 5 give an overview of the median computation times needed to solve the winner determination problem using these three approaches. Horizontally, the number of bidders n varies from 5 to 100, while the number of

Table 4. Median computation times [s] for n bidders and m items using BOI

	$n = 5$	10	25	50	75	100
$m = 5$	0.000	0.010	0.010	0.020	0.030	0.040
10	0.015	0.020	0.055	0.055	0.105	0.130
25	0.480	0.460	0.760	2.445	9.480	16.825
50	20.855	29.105	45.605	129.870	227.370	353.970

Table 5. Median computation times [s] for n bidders and m items using B&C

	$n = 5$	10	25	50	75	100
$m = 5$	0.000	0.010	0.010	0.020	0.030	0.040
10	0.015	0.020	0.040	0.055	0.100	0.130
25	0.485	0.495	0.815	2.445	6.790	13.605
50	20.855	29.215	37.970	129.870	238.785	514.370

Table 6. Median computation times [s] for n bidders and m items using BOS

	$n = 5$	10	25	50	75	100
$m = 5$	0.020	0.025	0.040	0.050	0.070	0.105
10	0.040	0.060	0.140	0.260	0.535	0.740
25	0.530	1.235	3.245	10.595	18.120	28.960
50	14.665	22.615	73.035	191.670	350.340	589.940

items m ranges from 5 to 50 vertically. All computation times are expressed in seconds.

As could be expected, the computation time is determined more by the number of items in the auction, than by the number of bidders. All instances with up to 10 items are solved in less than a second by all algorithms; here the branch-and-price algorithms clearly perform better. Auctions with 50 items are also solved in less than 20 minutes on average by all algorithms. For these instances, average computation times of the branch-and-price algorithms are often higher than the computation times of the branch-and-cut algorithm, while median computation times are often lower. Thus, we conclude that most instances are solved faster by the branch-and-price algorithms, however, for a few instances branch-and-price takes much more time than branch-and-cut.

Finally, Table 7 gives an overview of the performance details of the three algorithms. Column A gives the average number of nodes in the branching tree that were explored. Column B represents the average number of pricing rounds, and column C gives the average number of variables that were generated (these columns are not applicable for the branch-and-cut algorithm). On the rows, we find the instances, where the first number indicates the number of items and the second gives the number of bidders. There seems to be no systematic difference between the branch-and-price algorithms for any of the three parameters described in this table. The branch-and-cut algorithm solves very little nodes in its branching tree, compared to the branch-and-price algorithms. In many cases,

Table 7. Performance details for the three algorithms (BOI, BOS, B&C)

	BOI			**BOS**			**B&C**
Inst.	A	B	C	A	B	C	A
5-5	2.2	3.9	33.3	2.4	4.7	34.4	1.0
5-10	1.3	3.7	68.4	1.4	3.4	68.3	1.0
5-25	2.5	4.5	142.2	1.6	3.2	141.6	1.0
5-50	1.3	3.2	266.9	1.2	3.1	266.9	1.0
5-75	1.0	2.4	435.9	1.0	2.4	435.9	1.0
5-100	1.0	2.0	565.0	1.0	2.0	565.0	1.0
10-5	7.6	18.3	124.2	9.4	27.5	93.7	1.2
10-10	7.6	16.5	201.8	5.4	16.8	193.8	1.5
10-25	4.9	10.2	363.8	2.8	8.4	352.0	1.0
10-50	3.7	8.0	788.9	1.8	5.4	782.2	1.0
10-75	2.3	6.9	1,117.7	1.6	6.0	1,113.8	1.0
10-100	2.3	7.0	1,459.5	2.6	8.4	1,455.4	1.0
25-5	7.7	72.3	1,017.7	6.4	89.4	723.1	1.2
25-10	2.0	37.8	990.7	6.8	51.7	864.5	1.5
25-25	4.4	31.2	1,793.3	3.8	33.2	1,752.1	1.0
25-50	8.6	59.7	3,703.2	5.2	55.1	3,602.4	1.0
25-75	30.0	123.5	5,402.7	32.8	143.1	5,412.7	1.0
25-100	96.8	349.3	7,564.0	163.0	635.3	7,895.2	1.3
50-5	21.4	3,095.8	4,745.9	37.9	1,279.2	2,872.6	11.8
50-10	12.2	592.6	3,963.3	27.9	611.7	4,112.8	1.0
50-25	315.1	1,494.0	11,752.4	1,029.6	2,806.7	10,141.0	1.2
50-50	361.6	938.5	16,278.1	67.6	468.4	14,359.3	1.0
50-75	102.5	828.5	20,773.4	106.7	852.0	20,776.0	1.0
50-100	96.0	839.5	30,538.0	100.4	995.4	31,120.3	5.7

the branch-and-cut algorithm prefers generating valid inequalities in the root node to branching.

References

1. Bleischwitz, Y., Kliewer, G.: Accelerating Vickrey payment computation in combinatorial auctions for an airline alliance. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 228–239. Springer, Heidelberg (2005)
2. Day, R.W.: Expressing preferences with price-vector agents in combinatorial auctions. PhD thesis, University of Maryland (2004)
3. Day, R.W., Raghavan, S.: Matrix bidding in combinatorial auctions. Manuscript (2006)
4. Goossens, D.: Exact methods for combinatorial auctions. PhD thesis, K.U.Leuven (2006)

5. Goossens, D., Maas, A.J.T., Spieksma, F.C.R., van de Klundert, J.J.: Exact algorithms for procurement problems under a total quantity discount structure. European Journal of Operational Research 178(2), 603–626 (2007)
6. Grötschel, M.L., Lovász, L., Schrijver, A.: Geometric Algorithms and Combinatorial Optimization. Springer, Heidelberg (1988)
7. Hoos, H., Boutilier, C.: Solving combinatorial auctions using stochastic local search. In: AAAI/IAAI '00: Proceedings of the 17th national conference on artificial intelligence and 12th conference on innovative applications of artificial intelligence, Austin, USA, AAAI Press/The MIT Press, pp. 22–29 (2000)
8. Kann, V.: Maximum bounded 3-dimensional matching is max snp-complete. Information Processing Letters 37(1), 27–35 (1991)
9. Leyton-Brown, K., Shoham, Y., Tennenholtz, M.: An algorithm for multi-unit combinatorial auctions. In: AAAI/IAAI '00: Proceedings of the 17th national conference on artificial intelligence and 12th conference on innovative applications of artificial intelligence, Austin, USA, AAAI Press / The MIT Press, pp. 56–61 (2000)
10. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Empirical hardness models for combinatorial auctions. In: Cramton, P., Steinberg, R., Shoham, Y. (eds.) Combinatorial auctions, pp. 479–504. MIT Press, Cambridge (2005)
11. Leyton-Brown, K., Shoham, Y.: A test suite for combinatorial auctions. In: Cramton, P., Steinberg, R., Shoham, Y. (eds.) Combinatorial auctions, pp. 451–478. MIT Press, Cambridge (2005)
12. Nisan, N.: Bidding and allocation in combinatorial auctions. In: EC '00. Proceedings of the 2nd ACM conference on electronic commerce, Minneapolis, USA, pp. 1–12. ACM Press, New York (2000)
13. Parkes, D.C.: iBundle: an efficient ascending price bundle auction. In: EC '99. Proceedings of the ACM conference on electronic commerce, Denver, USA, pp. 148–157. ACM Press, New York (1999)
14. Petrank, E.: The hardness of approximation: gap location. Computational Complexity 4(2), 133–157 (1994)
15. Rothkopf, M., Pekeç, A., Harstad, R.M.: Computationally manageable combinational auctions. Management Science 44(8), 1131–1147 (1998)
16. Ryan, D., Foster, B.: An integer programming approach to scheduling. In: Wren, A. (ed.) Computer scheduling of public transport: urban passenger vehicle and crew scheduling. North-Holland, pp. 269–280 (1981)
17. Sandholm, T.: Algorithm for optimal winner determination in combinatorial auctions. Artificial Intelligence 135(1-2), 1–54 (2002)
18. Sandholm, T., Suri, S., Gilpin, A., Levine, D.: CABOB: a fast optimal algorithm for combinatorial auctions. Management Science 51, 374–390 (2005)
19. Vanderbeck, F., Wolsey, L.: An exact algorithm for ip column generation. Operations Research Letters 19, 151–159 (1996)
20. Van Hoesel, S., Müller, R.: Optimization in electronic markets: examples in combinatorial auctions. Netnomics 3(1), 23–33 (2001)

Author Index

Althaus, Ernst 420
Asgeirsson, Eyjolfur 285

Babenko, Maxim 256
Baumann, Tobias 420
Bazgan, Cristina 406
Behle, Markus 379
Betzler, Nadja 297

Chroni, Maria 189
Cortes, Corinna 1

Deineko, Vladimir 136
Delling, Daniel 52
Derryberry, Jonathan 256
Dixit, Kashyap 217
Drummond, Lúcia M.A. 175

Eisenbrand, Friedrich 338
Erten, Cesim 122

Fanelli, Angelo 324
Farshi, Mohammad 270
Figueroa, Karina 229
Flammini, Michele 324
Fredriksson, Kimmo 203, 229

Gerlach, Wolfgang 217
Goldberg, Andrew V. 38, 256
Goodrich, Michael T. 94
Goossens, Dries R. 433
Gudmundsson, Joachim 270
Guo, Yan 352

Hüffner, Falk 297
Hugot, Hadrien 406

Jünger, Michael 379

Kaplan, Haim 38
Karataş, Ömer 122
Karrenbauer, Andreas 338
Köhler, Ekkehard 365
Koukopoulos, Dimitrios 189

Lawson, Andrew 352
Liebchen, Christian 365
Liers, Frauke 379

Mäkinen, Veli 217
Mango, Domenico 324
Maue, Jens 242
Melideo, Giovanna 324
Mohri, Mehryar 1
Moscardelli, Luca 324

Niedermeier, Rolf 297
Nikitin, Fedor 203
Nikoletseas, Sotiris 161
Nikolopoulos, Stavros D. 189

Papamanthou, Charalampos 94
Pessoa, Artur 150
Powell, Olivier 161
Putze, Felix 108
Poggi de Aragão, Marcus 150

Rastogi, Ashish 1
Reich, Alexander 365
Ribeiro, Celso C. 393
Rizzi, Romeo 365

Sanders, Peter 23, 66, 108, 242
Santos, Marcelo 175
Schömer, Elmar 420
Schultes, Dominik 23, 66
Serna, Maria 37
Singler, Johannes 108
Smorodkina, Ekaterina 311
Sözdinler, Melih 122
Spieksma, Frits C.R. 433
Stein, Cliff 285

Tamassia, Roberto 94
Tang, Jijun 352
Tarjan, Robert 256
Tarjan, Robert E. 80
Tauritz, Daniel 311
Thakur, Mayur 311
Tiskin, Alexander 136
Toso, Rodrigo F. 393

Uchoa, Eduardo 150, 175

Välimäki, Niko 217
Vanderpooten, Daniel 406

Wagner, Dorothea 52
Werneck, Renato F. 38, 80
Werth, Kai 420
Wünsch, Gregor 365

Xu, Chihao 338

Ye, Fei 352

Zhou, Yunhong 256