LNCS 6630

Panos M. Pardalos
Steffen Rebennack (Eds.)

Experimental Algorithms

10th International Symposium, SEA 2011
Kolimpari, Chania, Crete, Greece, May 2011
Proceedings

Lecture Notes in Computer Science 6630

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

Alfred Kobsa
University of California, Irvine, CA, 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
TU Dortmund University, Germany

Madhu Sudan
Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Panos M. Pardalos Steffen Rebennack (Eds.)

Experimental Algorithms

10th International Symposium, SEA 2011
Kolimpari, Chania, Crete, Greece, May 5-7, 2011
Proceedings

Springer

Volume Editors

Panos M. Pardalos
303 Weil Hall
P.O. Box 116595
Gainesville, FL 32611-6595
E-mail: pardalos@ise.ufl.edu

Steffen Rebennack
Colorado School of Mines
Division of Economics and Business
Engineering Hall 310, 816 15th Street, Golden, CO 80401, USA
E-mail: srebenna@mines.edu

ISSN 0302-9743
e-ISSN 1611-3349
ISBN 978-3-642-20661-0
e-ISBN 978-3-642-20662-7
DOI 10.1007/978-3-642-20662-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011925898

CR Subject Classification (1998): F.2, I.2, H.3-4, F.1, C.2, D.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2011
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.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

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

Printed on acid-free paper

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

Preface

The 10th International Symposium on Experimental Algorithms (SEA 2011) was held in Kolimpari (Chania, Greece), during May 5–7, 2011. Previously known as the Workshops on Experimental Algorithms (WEA), the WEA/SEA Symposia are intended to be an international forum for researchers in the area of design, analysis, and experimental evaluation and engineering of algorithms, as well as in various aspects of computational optimization and its applications.

Past symposia were held in Ischia (2010), Dortmund (2009), Cape Cod (2008), Rome (2007), Menorca (2006), Santorini (2005), Rio de Janeiro (2004), Monte Verita (2003), and Riga (2002).

The present volume contains contributed papers accepted for publication, and papers based on three invited plenary talks. All papers underwent a strict refereeing process. From a total of 83 submitted contributed papers, 36 were selected for this special volume of *Lecture Notes in Computer Science* (LNCS). An additional nine contributed papers were accepted for a special issue of the Springer journal *Optimization Letters*. Thus, the 2011 International Symposium on Experimental Algorithms had an acceptance rate of 54%.

We would like to take this opportunity to thank all members of the Program Committee, the authors who submitted their papers, and the referees for their hard work. The quality of the aceppted papers was significantly influenced by constructive critical reviews. High-quality papers are always an important factor in maintaining and establishing a strong conference program.

We would like to thank Springer for publishing our proceedings in their well-known book series LNCS and for their support. In addition, we would like to thank the staff at the Orthodox Academy of Crete (OAC) for their help and hospitality. The location of OAC, next to the sea with a majestic view of the beautiful city of Chania, was an inspiration for a scientific gathering such as SEA 2011!

Finally, we would like to express our sincere thanks to the Steering Committee for providing us with the opportunity to serve as Program Chairs of SEA 2011 and for the responsibilities of selecting the Program Committee, the conference program, and publications. We are happy with the excellent topics presented at the conference and we look forward to SEA 2012.

May 2011

Panos M. Pardalos
Steffen Rebennack

Table of Contents

Experimental Algorithms and Applications

Invited Papers

Approximability of Symmetric Bimatrix Games and Related Experiments . . . 1
Spyros Kontogiannis and Paul Spirakis

Metaheuristic Optimization: Algorithm Analysis and Open Problems . . . 21
Xin-She Yang

Contributed Papers

Convexity and Optimization of Condense Discrete Functions . . . 33
Emre Tokgöz, Sara Nourazari, and Hillel Kumin

Path Trading: Fast Algorithms, Smoothed Analysis, and Hardness Results . . . 43
André Berger, Heiko Röglin, and Ruben van der Zwaan

Hierarchical Delaunay Triangulation for Meshing . . . 54
Shu Ye and Karen Daniels

A Parallel Multi-start Search Algorithm for Dynamic Traveling Salesman Problem . . . 65
Weiqi Li

Online Dictionary Matching with Variable-Length Gaps . . . 76
Tuukka Haapasalo, Panu Silvasti, Seppo Sippu, and Eljas Soisalon-Soininen

Dynamic Arc-Flags in Road Networks . . . 88
Gianlorenzo D'Angelo, Daniele Frigioni, and Camillo Vitale

Efficient Routing in Road Networks with Turn Costs . . . 100
Robert Geisberger and Christian Vetter

On Minimum Changeover Cost Arborescences . . . 112
Giulia Galbiati, Stefano Gualandi, and Francesco Maffioli

Localizing Program Logical Errors Using Extraction of Knowledge from Invariants . . . 124
Mojtaba Daryabari, Behrouz Minaei-Bidgoli, and Hamid Parvin

Compressed String Dictionaries 136
Nieves R. Brisaboa, Rodrigo Cánovas, Francisco Claude, Miguel A. Martínez-Prieto, and Gonzalo Navarro

Combinatorial Optimization for Weighing Matrices with the Ordering Messy Genetic Algorithm 148
Christos Koukouvinos and Dimitris E. Simos

Improved Automated Reaction Mapping 157
Tina Kouri and Dinesh Mehta

An Experimental Evaluation of Incremental and Hierarchical k-Median Algorithms 169
Chandrashekhar Nagarajan and David P. Williamson

Engineering the Modulo Network Simplex Heuristic for the Periodic Timetabling Problem 181
Marc Goerigk and Anita Schöbel

Practical Compressed Document Retrieval 193
Gonzalo Navarro, Simon J. Puglisi, and Daniel Valenzuela

Influence of Pruning Devices on the Solution of Molecular Distance Geometry Problems 206
Antonio Mucherino, Carlile Lavor, Therese Malliavin, Leo Liberti, Michael Nilges, and Nelson Maculan

An Experimental Evaluation of Treewidth at Most Four Reductions 218
Alexander Hein and Arie M.C.A. Koster

A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks 230
Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck

Hierarchy Decomposition for Faster User Equilibria on Road Networks 242
Dennis Luxen and Peter Sanders

Matching in Bipartite Graph Streams in a Small Number of Passes 254
Lasse Kliemann

Beyond Unit Propagation in SAT Solving 267
Michael Kaufmann and Stephan Kottler

Designing Difficult Office Space Allocation Problem Instances with Mathematical Programming 280
Özgür Ülker and Dario Landa-Silva

Speed Dating: An Algorithmic Case Study Involving Matching and Scheduling 292
Bastian Katz, Ignaz Rutter, Ben Strasser, and Dorothea Wagner

Experimental Evaluation of Algorithms for the Orthogonal Milling Problem with Turn Costs 304
Igor R. de Assis and Cid C. de Souza

A Branch-Cut-and-Price Algorithm for the Capacitated Arc Routing Problem 315
Rafael Martinelli, Diego Pecin, Marcus Poggi, and Humberto Longo

A Biased Random Key Genetic Algorithm Approach for Unit Commitment Problem 327
Luís A.C. Roque, Dalila B.M.M. Fontes, and Fernando A.C.C. Fontes

A Column Generation Approach to Scheduling of Periodic Tasks 340
Ernst Althaus, Rouven Naujoks, and Eike Thaden

Fuzzy Clustering the Backward Dynamic Slices of Programs to Identify the Origins of Failure 352
Saeed Parsa, Farzaneh Zareie, and Mojtaba Vahidi-Asl

Listing All Maximal Cliques in Large Sparse Real-World Graphs 364
David Eppstein and Darren Strash

Customizable Route Planning 376
Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck

Efficient Algorithms for Distributed Detection of Holes and Boundaries in Wireless Networks 388
Dennis Schieferdecker, Markus Völker, and Dorothea Wagner

Explanations for the Cumulative Constraint: An Experimental Study 400
Stefan Heinz and Jens Schulz

GRASP with Path-Relinking for Data Clustering: A Case Study for Biological Data 410
Rafael M.D. Frinhani, Ricardo M.A. Silva, Geraldo R. Mateus, Paola Festa, and Mauricio G.C. Resende

An Iterative Refinement Algorithm for the Minimum Branch Vertices Problem 421
Diego M. Silva, Ricardo M.A. Silva, Geraldo R. Mateus, José F. Gonçalves, Mauricio G.C. Resende, and Paola Festa

Generating Time Dependencies in Road Networks 434
Sascha Meinert and Dorothea Wagner

An Empirical Evaluation of Extendible Arrays 447
Stelios Joannou and Rajeev Raman

Author Index 459

Approximability of Symmetric Bimatrix Games and Related Experiments*

Spyros Kontogiannis[1,2] and Paul Spirakis[2]

[1] Dept. of Computer Science, University of Ioannina, 45110 Ioannina, Greece
kontog@cs.uoi.gr

[2] R.A. Computer Technology Institute, Patras Univ. Campus,
26504 Rio-Patra, Greece
spirakis@cti.gr

Abstract. In this work we present a simple quadratic formulation for the problem of computing Nash equilibria in *symmetric* bimatrix games, inspired by the well-known formulation of Mangasarian and Stone [26]. We exploit our formulation to shed light to the approximability of NE points. First we observe that any KKT point of this formulation (and indeed, any quadratic program) is also a stationary point, and vice versa. We then prove that *any* KKT point of the proposed formulation (is not necessarily itself, but) indicates a $\left(< \frac{1}{3}\right)-$NE point, which is polynomially tractable, given as input the KKT point. We continue by proposing an algorithm for constructing an $\left(\frac{1}{3}+\delta\right)-$NE point for any $\delta > 0$, in time polynomial in the size of the game and quasi-linear in $\frac{1}{\delta}$, exploiting Ye's algorithm for approximating KKT points of QPs [34]. This is (to our knowledge) the first polynomial time algorithm that constructs $\varepsilon-$NE points for symmetric bimatrix games for any ε close to $\frac{1}{3}$. We extend our main result to (asymmetric) win lose games, as well as to games with maximum aggregate payoff either at most 1, or at least $\frac{5}{3}$. To achieve this, we use a generalization of the Brown & von Neumann symmetrization technique [6] to the case of non-zero-sum games, which we prove that is approximation preserving. Finally, we present our experimental analysis of the proposed approximation and other quite interesting approximations for NE points in symmetric bimatrix games.

Keywords: Bimatrix games, indefinite quadratic optimization, Nash equilibrium approximation, symmetrization.

1 Introduction

One of the "holy grail quests" of theoretical computer science in the last decade, is the characterization of the computational complexity for constructing *any* Nash equilibrium (NE) in a finite normal form game. There has been a massive attack on various refinements of the problem (eg, a NE maximizing the payoff

* This work has been partially supported by the ICT Programme of the EU under contract number 258885 (SPITFIRE).

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 1–20, 2011.
© Springer-Verlag Berlin Heidelberg 2011

of some player, or its support size), that have lead to **NP**−hardness results (eg, [17,11]). Eventually the unconstrained problem proved to be **PPAD**−complete [12,15], even for the bimatrix case [8]. Even computing an $\left(n^{-\Theta(1)}\right)$−approximate NE for the bimatrix case is **PPAD**−complete [9], excluding even the existence of a FPTAS for the problem, unless **PPAD** = **P**. Additionally, it is well known that the celebrated algorithm of Lemke and Howson [24] may take an exponential number of steps to terminate [29].

Given the apparent hardness of computing NE in bimatrix games, two main research trends emerged quite naturally: To discover polynomial-time, constant-approximation algorithms (or even a PTAS) for the general case, or to identify general subclasses of games that admit a polynomial-time construction of *exact* NE, or at least a (F)PTAS. Even if one exchanges the "polynomiality" to "strict subexponentiality", there is still much room for research. Indeed, the first subexponential-time approximation scheme was provided in [25] (see also [3]), while a new one appeared only recently [33]. A sequence of papers have provided polynomial-time algorithms for various notions of approximation (eg, [20,13,5,14,32,22]), the current winners being the gradient-based algorithm of [32] that provides 0.3393−approximation for the most common notion of $\varepsilon-$*Nash equilibria*, and [22] that provides an LP-based 0.667−approximation for the more demanding notion of $\varepsilon-$*well supported approximate NE*.

As for exact solutions (or even FPTAS) for general subclasses of bimatrix games, it is well known (due to von Neumann's minimax theorem [27]) that any constant-sum bimatrix game is polynomial-time solvable. Trivially, any bimatrix game with a pure Nash equilibrium is also solvable in polynomial time. Finally, for the particular case of win-lose bimatrix games, [10] provided a linear-time (exact) algorithm for games with very sparse payoff matrices and [1] provided a polynomial-time algorithm when the graph of the game is planar. [19] introduced a hierarchy of the bimatrix games, according to the rank of the matrix $R + C$ of the game $\langle R, C\rangle$, which was called the *rank of the game*. Then, for any *fixed* constant $k \geq 0$, they present a FPTAS for bimatrix games of rank k. Note that rank−0 games are zero-sum, while for rank−1 games it was recently proved [2] that they are also polynomial-time solvable. [21] proposed a subclass of polynomial-time solvable bimatrix games, called *mutually concave* games. This class contains all constant-sum games but is much richer, and is incomparable to games of fixed rank, since even rank−1 games may not be mutually concave, and on the other hand one can easily construct mutually concave games which have full rank. In this work it was proved that these games are equivalent to the *strategically zero-sum games* [28]; a novel quadratic formulation for computing NE points in bimatrix games was also proposed which captures the NE approximability of the marginal distributions of (exact) correlated equilibria.

Our Contribution and Roadmap. In Section 3 we present a simple quadratic formulation for computing (exact) NE points of symmetric bimatrix games, which specializes the formulation of Mangasarian and Stone [26] to the symmetric case. We then prove that from any *given* KKT point of our formulation, we can construct in polynomial time a $\left(< \frac{1}{3}\right)$−NE point (Theorem 1).

In Section 4 we show how to construct approximate NE points with approximation ratio arbitrarily close to $\frac{1}{3}$, in polynomial time (Theorem 2). We also show that there exist even better approximate NE points which would be polynomial-time constructible, given *any* initial $\delta-$KKT point with a Lagrange dual that is also a $\delta-$KKT point, for sufficiently small $\delta > 0$ (Theorem 3). In Section 5 we extend our approach to asymmetric win lose games, and games with aggregate payoff either at most 1 or at least $\frac{5}{3}$. In particular, we first generalize the symmetrization method of Brown and von Neumann, and we then prove that it is indeed approximation preserving (Lemma 2). We exploit this fact to provide approximate NE points with ratio arbitrarily close to $\frac{1}{3}$ for these classes of (asymmetric) games (Theorem 4).

The last part of our paper concerns our experimental study of algorithms constructing approximate NE points in symmetric games. In Section 6 we explain all the algorithms that we consider, as well as the random game generator that we use. In Section 7 we summarize our observations on worst-case instances per approximation algorithm separately, as well as for hybrid approaches that always return the best approximation from a number of algorithms in each game. We close our discussion with some concluding remarks in Section 8.

2 Preliminaries

Algebraic Notation. We denote by $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ the sets of naturals, integers, rationals, and real numbers. The '+' subscript (eg, $\mathbb{R}_+$) implies the nonnegative subset of a set, and a '++' subscript (eg, $\mathbb{Z}_{++}$) refers to the positive subset of a set. For any $d \in \mathbb{Z}_{++}$, $[d] = \{1, 2, \ldots, d\}$. We denote by $\mathbb{R}^d$ ($\mathbb{R}^d_+$) the $d-$dimensional (non-negative) real space. For any real vector $\mathbf{z} \in \mathbb{R}^d$, let $\max(\mathbf{z}) \equiv \max_{i\in[d]}\{|z_i|\}$. For any matrix $A \in \mathbb{R}^{m\times n}$, and any $(i, j) \in [m] \times [n]$, $A_{i,j}$ is its $(i, j)-$th element, $A_{i,\star}$ is its $i-$th row (as a row vector) and $A_{\star,j}$ is its $j-$th column (as a column vector). A^T is the transpose matrix of A. We denote by $\mathbf{I}$ the identity matrix, by $\mathbf{E}$ the "all-ones" matrix and by $\mathbf{O}$ the "all-zeros" matrix (of proper dimensions). For any $A, B \in \mathbb{R}^{m\times n}$, $A{\cdot}B^T = AB^T$ is the $m\times m$ matrix that is produced by their inner product, $A{\circ}B = [A_{i,j} \cdot B_{i,j}]_{(i,j)\in[m]\times[n]}$ is the componentwise product, and $A{\bullet}B = \mathbf{1}^T A{\circ}B\mathbf{1}$.

For any $\varepsilon > 0$, $B(\mathbf{z}, \varepsilon) \equiv \{\mathbf{x} \in \mathbb{R}^d : \|\mathbf{x} - \mathbf{z}\| < \varepsilon\}$ and $\bar{B}(\mathbf{z}, \varepsilon) \equiv \{\mathbf{x} \in \mathbb{R}^d : \|\mathbf{x} - \mathbf{z}\| \le \varepsilon\}$ are the open and closed ball of radius ε around $\mathbf{z}$ respectively, where $\|\cdot\|$ denotes the Euclidean norm in $\mathbb{R}^d$. $\Delta_d = \{\mathbf{z} \in \mathbb{R}^d_+ : \mathbf{1}^T\mathbf{z} = 1\}$ is the space of $d-$dimensional probability vectors.

Game Theoretic Notation. For any $2 \le m \le n$, we denote by $\langle R, C\rangle$ an $m\times n$ **bimatrix game**, where the first player (aka the row player) has $R \in \mathbb{R}^{m\times n}$ as its payoff matrix and the second player (aka the column player) has $C \in \mathbb{R}^{m\times n}$ as its payoff matrix. If it happens that $C = R^T$ then we have a **symmetric** game. If $R, C \in \mathbb{Q}^{m\times n}$ then we have a **rational bimatrix game**. The subclass of rational games in which $R, C \in (\mathbb{Q}\cap[0, 1])^{m\times n}$ are called **normalized bimatrix games**. These are the games of concern in this work, for computational reasons. Finally, a game in which $R, C \in \{0, 1\}^{m\times n}$ is called a **win-lose game**.

The row (column) player chooses one of the rows (columns) of the payoff bimatrix $(R,C) = (R_{i,j}, C_{i,j})_{(i,j)\in[m]\times[n]}$ as her **action**. For any profile of actions $(i,j) \in [m]\times[n]$ for the two players, the payoff to the row (column) player is $R_{i,j}$ ($C_{i,j}$). A (mixed in general) **strategy** for the row (column) player is a probability distribution $\mathbf{x} \in \Delta_m$ ($\mathbf{y} \in \Delta_n$), according to which she determines her action, independently of the opponent's final choice of action. If all the probability mass of a strategy is assigned to a particular action of the corresponding player, then we refer to a **pure strategy**. The **utility** of the row (column) player for a **profile of strategies** $(\mathbf{x},\mathbf{y})$ is the expected payoff $\mathbf{x}^T R\mathbf{y} = \sum_{i\in[m]}\sum_{j\in[n]} R_{i,j}x_i y_j$ ($\mathbf{x}^T C\mathbf{y}$) that she gets. For any real number $\varepsilon \geq 0$, a profile of strategies $(\overline{\mathbf{x}},\overline{\mathbf{y}}) \in \Delta_m \times \Delta_n$ is an $\varepsilon-$**Nash equilibrium** ($\varepsilon-$NE in short) of $\langle R,C\rangle$, iff each player's strategy is an approximate best response (within an *additive* term of ε) to the opponent's strategy: $\forall \mathbf{x} \in \Delta_m, \overline{\mathbf{x}}^T R\overline{\mathbf{y}} \geq \mathbf{x}^T R\overline{\mathbf{y}} - \varepsilon$ and $\forall \mathbf{y} \in \Delta_n, \overline{\mathbf{x}}^T C\overline{\mathbf{y}} \geq \overline{\mathbf{x}}^T C\mathbf{y} - \varepsilon$. We denote by $NE(\varepsilon,R,C)$ the set of all these points of $\langle R,C\rangle$. We refer to a **symmetric profile** if both players adopt exactly the same strategy. The $\varepsilon-$NE points corresponding to symmetric profiles are called **symmetric** $\varepsilon-$**Nash equilibria** ($\varepsilon-$SNE points in short). For any profile $(\mathbf{x},\mathbf{y}) \in \Delta_m \times \Delta_n$, the **regrets** of the row and the column player in $\langle R,C\rangle$ are defined as $R_I(\mathbf{x},\mathbf{y}) = \max(R\mathbf{y}) - \mathbf{x}^T R\mathbf{y}$ and $R_{II}(\mathbf{x},\mathbf{y}) = \max(C^T\mathbf{x}) - \mathbf{x}^T C\mathbf{y}$ respectively.

Quadratic Programming. Consider the following quadratic program:

$$\text{(QP)} \qquad \boxed{\text{minimize } \left\{ f(\mathbf{z}) = \mathbf{c}^T\mathbf{z} + \tfrac{1}{2}\mathbf{z}^T D\mathbf{z} : \; A\mathbf{z} \geq \mathbf{b}; \; C\mathbf{z} = \mathbf{d} \right\}}$$

where $D \in \mathbb{R}^{n\times n}$ is a *symmetric* real matrix, $A \in \mathbb{R}^{m\times n}$, $C \in \mathbb{R}^{k\times n}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{d} \in \mathbb{R}^k$. We denote by $feas(QP) = \{\mathbf{z} \in \mathbb{R}^n : A\mathbf{z} \geq \mathbf{b};\; C\mathbf{z} = \mathbf{d}\}$ the set of all feasible points for (QP). Additionally, consider the following (possibly empty) subsets of feasible points:

- **(Globally) Optimal Points:** The points with the minimum objective value: $opt(QP) = \{\mathbf{z}^* \in feas(QP) : \forall \mathbf{z} \in feas(QP),\; f(\mathbf{z}) \geq f(\mathbf{z}^*)\}$.
- **Locally Optimal Points:** The points with the minimum objective value in an open ball around them. Ie:
  $loc(QP) = \{\underline{\mathbf{z}} \in feas(QP) : \exists \varepsilon > 0, \forall \mathbf{z} \in feas(QP) \cap B(\underline{\mathbf{z}},\varepsilon), f(\mathbf{z}) \geq f(\underline{\mathbf{z}})\}$.
- $\delta-$**KKT Points:** $\forall \delta \geq 0$, $kkt(\delta,QP)$ is the set of feasible points $\overline{\mathbf{z}} \in feas(QP)$ for which there exists a vector $(\overline{\lambda},\overline{\mu})$ of Lagrange multipliers satisfying the following (approximate) KKT conditions (named after Karush, Kuhn and Tucker, cf. [23, Theorem 3.3] and [4, Section 3.3.1]) at those points:
  $$\begin{aligned} &\textit{Stationarity:} && D\overline{\mathbf{z}} + \mathbf{c} = A^T\overline{\lambda} + C^T\overline{\mu} \\ &\textit{(Approximate) Complementarity:} && 0 \leq \overline{\lambda}^T \cdot (A\overline{\mathbf{z}} - \mathbf{b}) \leq \delta \\ &\textit{Feasibility:} && \overline{\lambda} \geq \mathbf{0},\; A\overline{\mathbf{z}} \geq \mathbf{b},\; C\overline{\mathbf{z}} = \mathbf{d} \end{aligned}$$
  We denote the subset $kkt(0,QP)$ of exact KKT points as $kkt(QP)$.
- $\delta-$**Stationary Points:** $\forall \delta \geq 0$, $sta(\delta,QP) \subseteq feas(QP)$ is the set of points satisfying (approximately) a stationarity property known as **Fermat's Rule**:
  $sta(\delta,QP) = \{\overline{\mathbf{z}} \in feas(QP) : \forall \mathbf{z} \in feas(QP), \nabla f(\overline{\mathbf{z}})^T \cdot (\mathbf{z} - \overline{\mathbf{z}}) \geq -\delta\}$.
  We denote the subset $sta(0,QP)$ of exact stationary points as $sta(QP)$.

A well-known property of quadratic programs with linear constraints is that $kkt(QP) = sta(QP)$ (cf. [23, Theorems 3.4 & 3.5] and the comment on this issue between these two theorems). This is a quite interesting property in our case of NE points in bimatrix games, since it essentially assures that the stationary points which are targeted by the descent method of Tsaknakis and Spirakis [32] are indeed the KKT points of the quadratic formulation that we shall propose shortly (for symmetric games), or the formulation of Mangasarian and Stone [26] (for general games).

3 Approximability of NE Points via (exact) KKT Points

In this section we provide, for any normalized (rational) matrix $S \in [0,1]^{n\times n}$, a quadratic program (SMS) for which it holds that $opt(SMS) = NE(0, S, S^T)$. We then prove that any KKT point of (SMS) indicates a profile that is a $\left(< \frac{1}{3}\right)-$NE of the symmetric game. Additionally, given a KKT point, such a profile can be constructed in polynomial time.

Two crucial observations for our formulations are the following: (i) Any symmetric bimatrix game $\langle A, B\rangle = \langle S, S^T\rangle$ has at least one SNE point. (ii) Any symmetric profile assures not only the same payoffs to both players, but also the same payoff vectors against the opponent (thus, both players have the same regrets in case of symmetric profiles): $\forall S \in [0,1]^{n\times n}, \forall \mathbf{z} \in \Delta_n$,

$$\begin{aligned} &\textit{common payoffs}: && \mathbf{z}^T B\mathbf{z} = \mathbf{z}^T S^T \mathbf{z} = \mathbf{z}^T S\mathbf{z} = \mathbf{z}^T A\mathbf{z} \\ &\textit{common payoff vectors}: && A\mathbf{z} = S\mathbf{z} = (S^T)^T\mathbf{z} = B^T\mathbf{z} \end{aligned}$$

The quadratic program that we use is the following adaptation of Mangasarian and Stone's program (for another quadratic program whose set of global optima has a bijective map to the set of NE points of a bimatrix game, see [21]):

$$\text{(SMS)} \quad \boxed{\begin{array}{llll} \textbf{minimize } f(s,\mathbf{z}) = s - \mathbf{z}^T S\mathbf{z} = s - \frac{1}{2}\mathbf{z}^T Q\mathbf{z} \\ \text{subject to:} \quad -\mathbf{1}s + \quad S\mathbf{z} \quad \leq \mathbf{0} \\ \qquad\qquad\qquad - \quad \mathbf{1}^T\mathbf{z} + 1 = 0 \\ \qquad\qquad s \in \mathbb{R}, \quad \mathbf{z} \in \mathbb{R}^n_+ \end{array}}$$

where $Q = S + S^T$ is a *symmetric* $n \times n$ real matrix. Observe that any probability distribution $\mathbf{z} \in \Delta_n$ induces the feasible solution $(s = \max(S\mathbf{z}), \mathbf{z})$ of (SMS). Moreover, $f(s,\mathbf{z})$ is an upper bound on the common regret of the two players wrt the symmetric profile $(\mathbf{z},\mathbf{z})$: $f(s,\mathbf{z}) \geq R(\mathbf{z}) = \max(S\mathbf{z}) - \mathbf{z}^T S\mathbf{z}$. Therefore, the objective value of a feasible point in (SMS) is non-negative and may reach zero only for SNE points of $\langle S, S^T\rangle$. In the sequel, for any probability vector $\mathbf{z} \in \Delta_n$, we shall overload notation and use $f(\mathbf{z})$ to denote the value $f(\max(S\mathbf{z}), \mathbf{z})$, for sake of simplicity.

The conditions determining a $\delta-$KKT point $(\bar{s}, \overline{\mathbf{z}})$ of (SMS) (for any fixed $\delta \geq 0$) along with its Lagrange vector $(\overline{\mathbf{w}} \in \mathbb{R}^n_+, \bar{\zeta} \in \mathbb{R}, \overline{\mathbf{u}} \in \mathbb{R}^n_+)$ are the following:

(KKTSMS)

Stationarity	$\nabla f(\bar{s}, \overline{\mathbf{z}}) = \binom{1}{-S\overline{\mathbf{z}}-S^T\overline{\mathbf{z}}} = \binom{\mathbf{1}^T\overline{\mathbf{w}}}{-S^T\overline{\mathbf{w}}+\overline{\mathbf{u}}+\mathbf{1}\bar{\zeta}}$	(1)
(Approximate) Complementarity	$0 \leq \binom{\overline{\mathbf{w}}}{\overline{\mathbf{u}}}^T \cdot \binom{\mathbf{1}\bar{s}-S\overline{\mathbf{z}}}{\overline{\mathbf{z}}} \leq \delta$	(2)
Primal Feasibility	$\bar{s} \in \mathbb{R},\ S\overline{\mathbf{z}} \leq \mathbf{1}\bar{s},\ \mathbf{1}^T\overline{\mathbf{z}} = 1,\ \overline{\mathbf{z}} \geq \mathbf{0}$	(3)
Dual Feasibility	$\overline{\mathbf{w}} \geq \mathbf{0},\ \bar{\zeta} \in \mathbb{R},\ \overline{\mathbf{u}} \geq \mathbf{0}$	(4)

Observe that the Lagrange vector $\overline{\mathbf{w}}$ is a probability distribution (eq. (1)), which is an approximate best response of the row player against strategy $\overline{\mathbf{z}}$ of the column player (2): The payoff vector $S\overline{\mathbf{z}}$ of the row player is upper bounded by $\mathbf{1}\bar{s}$ (eq. (3)). Therefore, $\overline{\mathbf{u}}^T\overline{\mathbf{z}} + \max(S\overline{\mathbf{z}}) - \overline{\mathbf{w}}^T S\overline{\mathbf{z}} \leq \overline{\mathbf{u}}^T\overline{\mathbf{z}} + \bar{s} - \overline{\mathbf{w}}^T S\overline{\mathbf{z}} \leq \delta \Rightarrow R_I(\overline{\mathbf{w}}, \overline{\mathbf{z}}) + \overline{\mathbf{u}}^T\overline{\mathbf{z}} \leq \delta$.

It is mentioned at this point that since our objective function has values in $[0,1]$, which can be actually attained (any SNE point indicates a solution of (SMS) of zero regret, and in worst case the regret may reach the value of 1 since the game is normalized), our notion of approximate KKT points is identical to the one considered in Ye's work [34]. Therefore, a $\delta-$KKT point of (SMS) may be constructed in time $\mathcal{O}\left(\left[\frac{n^6}{\delta}\log\left(\frac{1}{\delta}\right) + n^4\log(n)\right] \cdot \left[\log\log\left(\frac{1}{\delta}\right) + \log(n)\right]\right)$. The next lemma proves a fundamental property of any *exact* KKT point of (SMS):

Lemma 1. *For any $m, n \geq 2,\ S \in [0,1]^{m\times n}$, every $(\max(S\overline{\mathbf{z}}), \overline{\mathbf{z}}) \in kkt(SMS)$ and its associated Lagrange vector $(\overline{\mathbf{w}}, \overline{\mathbf{u}}, \bar{\zeta})$ satisfy the following properties: (i) $\bar{\zeta} = f(\overline{\mathbf{z}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{z}}$. (ii) $2f(\overline{\mathbf{z}}) = \overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{w}}^T\overline{\mathbf{u}}$. (iii) $2f(\overline{\mathbf{z}}) + f(\overline{\mathbf{w}}) = R_I(\overline{\mathbf{z}}, \overline{\mathbf{w}}) - \overline{\mathbf{w}}^T\overline{\mathbf{u}}$.*

Proof. From the stationarity conditions (eq. (1)) the following holds:

$$
\begin{aligned}
& -S\overline{\mathbf{z}} - S^T\overline{\mathbf{z}} = -S^T\overline{\mathbf{w}} + \overline{\mathbf{u}} + \mathbf{1}\bar{\zeta} \\
\Rightarrow & \begin{cases} -\overline{\mathbf{z}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S^T\overline{\mathbf{z}} = -\overline{\mathbf{z}}^T S^T\overline{\mathbf{w}} + \underbrace{\overline{\mathbf{z}}^T\overline{\mathbf{u}}}_{=0} + \underbrace{\overline{\mathbf{z}}^T\mathbf{1}}_{=1}\bar{\zeta} \\ -\overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{w}}^T S^T\overline{\mathbf{z}} = -\overline{\mathbf{w}}^T S^T\overline{\mathbf{w}} + \overline{\mathbf{w}}^T\overline{\mathbf{u}} + \underbrace{\overline{\mathbf{w}}^T\mathbf{1}}_{=1}\bar{\zeta} \end{cases} \\
\Rightarrow & \begin{cases} \bar{\zeta} = -2\overline{\mathbf{z}}^T S\overline{\mathbf{z}} + \overline{\mathbf{z}}^T S^T\overline{\mathbf{w}} = f(\overline{\mathbf{z}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} \\ \bar{\zeta} = -\overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} + \overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \overline{\mathbf{w}}^T\overline{\mathbf{u}} \end{cases} \\
\Rightarrow & \begin{cases} \bar{\zeta} = f(\overline{\mathbf{z}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} \\ 2f(\overline{\mathbf{z}}) = -2\overline{\mathbf{z}}^T S\overline{\mathbf{z}} + 2\overline{\mathbf{w}}^T S\overline{\mathbf{z}} = -\overline{\mathbf{z}}^T S\overline{\mathbf{w}} + \overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \overline{\mathbf{w}}^T\overline{\mathbf{u}} \end{cases}
\end{aligned}
$$

Adding $f(\overline{\mathbf{w}}) = \max(S\overline{\mathbf{w}}) - \overline{\mathbf{w}}^T S\overline{\mathbf{w}}$ to both sides of the last expression, we get:

$$
\begin{aligned}
& 2f(\overline{\mathbf{z}}) + f(\overline{\mathbf{w}}) = \max(S\overline{\mathbf{w}}) - \overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} + \overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \overline{\mathbf{w}}^T\overline{\mathbf{u}} \\
\Rightarrow \quad & \boxed{3\min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} \leq 2f(\overline{\mathbf{z}}) + f(\overline{\mathbf{w}}) = R_I(\overline{\mathbf{z}}, \overline{\mathbf{w}}) - \overline{\mathbf{w}}^T\overline{\mathbf{u}}}
\end{aligned}
$$

Lemma 1 already assures that one of $(\overline{\mathbf{z}}, \overline{\mathbf{z}}), (\overline{\mathbf{w}}, \overline{\mathbf{w}})$ is a $\frac{1}{3}-$SNE point of $\langle S, S^T\rangle$, since $R_I(\overline{\mathbf{z}}, \overline{\mathbf{w}}) \leq 1$ (the payoff matrix is normalized) and $\overline{\mathbf{w}}^T\overline{\mathbf{u}} \geq 0$. We prove

next that *given* any $(\bar{s}, \overline{\mathbf{z}}) \in kkt(SMS)$, we can construct an $\left(< \frac{1}{3}\right)$ −SNE point in polynomial time.

Theorem 1. *For any given* $(\bar{s}, \overline{\mathbf{z}}) \in kkt(SMS)$, $\exists \varepsilon \in \left[0, \frac{1}{3}\right)$ *such that an* ε*−SNE point of the symmetric (normalized) bimatrix game* $\langle S, S^T \rangle$ *is polynomial-time constructible.*

Proof. Let $(\overline{\mathbf{w}}, \overline{\mathbf{u}}, \bar{\zeta})$ be the vector of the corresponding Lagrange multipliers in (KKTSMS) corresponding to $(\bar{s}, \overline{\mathbf{z}})$. They are polynomial-time computable, as the solution of the Linear Feasibility problem that evolves from(KKTSMS) for the fixed value of $(\bar{s} = \max(S\overline{\mathbf{z}}), \overline{\mathbf{z}})$. We have already proved in Lemma 1 that:

$$2f(\overline{\mathbf{z}}) + f(\overline{\mathbf{w}}) = \max(S\overline{\mathbf{w}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{w}}^T\overline{\mathbf{u}}$$

Clearly, if $f(\overline{\mathbf{z}}) \neq f(\overline{\mathbf{w}})$, then $\min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} < \frac{\max(S\overline{\mathbf{w}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{w}}^T\overline{\mathbf{u}}}{3} \leq \frac{1}{3}$ and we are done. So, let's assume that $f(\overline{\mathbf{z}}) = f(\overline{\mathbf{w}}) = \frac{1}{3}$. This in turn implies that:

$$\max(S\overline{\mathbf{w}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{w}}^T\overline{\mathbf{u}} = 1 \Rightarrow \boxed{\max(S\overline{\mathbf{w}}) = 1 \wedge \overline{\mathbf{z}}^T S\overline{\mathbf{w}} = 0 \wedge \overline{\mathbf{w}}^T\overline{\mathbf{u}} = 0}$$

since $\max(S\overline{\mathbf{w}}) \leq 1,\ \overline{\mathbf{z}}^T S\overline{\mathbf{w}} \geq 0,\ \overline{\mathbf{w}}^T\overline{\mathbf{u}} \geq 0$. From this we also deduce that:

$$\overline{\mathbf{w}}^T S\overline{\mathbf{w}} = -f(\overline{\mathbf{w}}) + \max(S\overline{\mathbf{w}}) \Rightarrow \boxed{\overline{\mathbf{w}}^T S\overline{\mathbf{w}} = \frac{2}{3}}$$

We now focus on $\overline{\mathbf{w}}$. If $(\max(S\overline{\mathbf{w}}), \overline{\mathbf{w}}) \notin kkt(SMS)$, then we may apply Ye's potential reduction algorithm (or any other polynomial-time algorithm that converges to a KKT point) with this as a starting point, and we shall get an approximate KKT point $(\max(S\hat{\mathbf{z}}), \hat{\mathbf{z}})$ having $f(\hat{\mathbf{z}}) < f(\overline{\mathbf{w}}) = \frac{1}{3}$. So, let's suppose that $(\max(S\overline{\mathbf{w}}), \overline{\mathbf{w}}) \in kkt(SMS)$. Let $(\tilde{\mathbf{w}}, \tilde{\mathbf{u}}, \tilde{\zeta})$ be the proper Lagrange multipliers for this new KKT point. By applying the stationarity conditions for $\overline{\mathbf{w}}$ this time, we have:

$$\begin{array}{c} 2f(\overline{\mathbf{w}}) + f(\tilde{\mathbf{w}}) = \max(S\tilde{\mathbf{w}}) - \overline{\mathbf{w}}^T S\tilde{\mathbf{w}} - \tilde{\mathbf{w}}^T\tilde{\mathbf{u}} \qquad \Rightarrow \\ \boxed{f(\tilde{\mathbf{w}}) = -\frac{2}{3} + \max(S\tilde{\mathbf{w}}) - \overline{\mathbf{w}}^T S\tilde{\mathbf{w}} - \tilde{\mathbf{w}}^T\tilde{\mathbf{u}}} \end{array}$$

Again, unless $f(\tilde{\mathbf{w}}) < \frac{1}{3}$ (in which case we would return $\tilde{\mathbf{w}}$), it must be the case that

$$\boxed{\max(S\tilde{\mathbf{w}}) = 1 \wedge \overline{\mathbf{w}}^T S\tilde{\mathbf{w}} = 0 \wedge \tilde{\mathbf{w}}^T\tilde{\mathbf{u}} = 0}$$

Since $(\max(S\overline{\mathbf{z}}), \overline{\mathbf{z}}), (\max(S\overline{\mathbf{w}}), \overline{\mathbf{w}}) \in kkt(SMS)$, we have:

$$\begin{array}{rcl} \left.\begin{array}{r} -S\overline{\mathbf{z}} - S^T\overline{\mathbf{z}} + S^T\overline{\mathbf{w}} = \overline{\mathbf{u}} + \mathbf{1}\bar{\zeta} \\ \bar{\zeta} = f(\overline{\mathbf{z}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} \end{array}\right\} & & \Rightarrow \\ -\tilde{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\tilde{\mathbf{w}} + \underbrace{\overline{\mathbf{w}}^T S\tilde{\mathbf{w}}}_{=0} = \tilde{\mathbf{w}}^T\overline{\mathbf{u}} + f(\overline{\mathbf{z}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} & & \Rightarrow \\ 0 \leq \tilde{\mathbf{w}}^T\overline{\mathbf{u}} = -\tilde{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\tilde{\mathbf{w}} - f(\overline{\mathbf{z}}) + \overline{\mathbf{z}}^T S\overline{\mathbf{z}} & & \Rightarrow \\ \boxed{\tilde{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} \leq -\frac{1}{3} - \overline{\mathbf{z}}^T S\tilde{\mathbf{w}} < 0} & & \end{array}$$

On the other hand:

$$\left.\begin{array}{r} -S\overline{\mathbf{w}} - S^T\overline{\mathbf{w}} + S^T\tilde{\mathbf{w}} = \tilde{\mathbf{u}} + \mathbf{1}\tilde{\zeta} \\ \tilde{\zeta} = f(\overline{\mathbf{w}}) - \overline{\mathbf{w}}^T S\overline{\mathbf{w}} = -\frac{1}{3} \end{array}\right\} \Rightarrow -\underbrace{\overline{\mathbf{z}}^T S\overline{\mathbf{w}}}_{=0} - \underbrace{\overline{\mathbf{z}}^T S^T\overline{\mathbf{w}}}_{=\max(S\overline{\mathbf{z}})} + \overline{\mathbf{z}}^T S^T\tilde{\mathbf{w}} = \overline{\mathbf{z}}^T\tilde{\mathbf{u}} - \frac{1}{3}$$

$$\Rightarrow 0 \le \overline{\mathbf{z}}^T\tilde{\mathbf{u}} = \frac{1}{3} - \max(S\overline{\mathbf{z}}) + \tilde{\mathbf{w}}^T S\overline{\mathbf{z}} \Rightarrow \underbrace{\max(S\overline{\mathbf{z}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{z}}}_{=f(\overline{\mathbf{z}})=\frac{1}{3}} \le \frac{1}{3} + \tilde{\mathbf{w}}S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}}$$

$$\Rightarrow \quad \boxed{\tilde{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} \ge 0}$$

which contradicts the previously stated inequality $\frac{1}{3} + \tilde{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} \le -\overline{\mathbf{z}}^T S\tilde{\mathbf{w}}$.

4 Polynomial Time Construction of $\left(\frac{1}{3} + \delta\right)$ −NE Points

The main point of our discussion in the previous section was that we considered as given an arbitrary exact KKT point of (SMS), from which we could then construct in polynomial time a $\left(< \frac{1}{3}\right)$ −SNE of the game. Therefore, we cannot yet claim that in overall this is a polynomial-time algorithm for constructing even an $\left(\frac{1}{3}\right)$ −NE point for symmetric bimatrix games, since the computation of an exact KKT point is hard. In order to achieve this, we have to restrict ourselves to approximate KKT points, which are indeed polynomial-time constructible (eg, via Ye's algorithm). The following theorem gives the answer to this question:

Theorem 2. *For any* rational *matrix $S \in [0,1]^{n\times n}$, and any $\delta > 0$, there exists an algorithm for constructing a $\left(\frac{1}{3} + \delta\right)$ −SNE of $\langle S, S^T\rangle$ in time polynomial in the size of the game and quasi-linear in δ.*

Proof. The argument is similar to the one of Theorem 1, but we now have to be more careful in handling approximate KKT points. We start by constructing (using Ye's algorithm) a $\delta-$KKT point of (SMS), $(\bar{s}, \overline{\mathbf{z}})$, along with its Lagrange multipliers $(\overline{\mathbf{w}}, \overline{\mathbf{u}}, \bar{\zeta})$. Wlog we can assume that $\bar{s} = \max(S\overline{\mathbf{z}})$. Clearly it holds that $\overline{\mathbf{w}}^T S\overline{\mathbf{z}} \le \max(S\overline{\mathbf{z}}) \le \overline{\mathbf{w}}^T S\overline{\mathbf{z}} + \delta \Rightarrow \overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} \le f(\overline{\mathbf{z}}) \le \overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} + \delta$. From the proof of Lemma 1 we know that

i. $\bar{\zeta} = \overline{\mathbf{w}}^T S\overline{\mathbf{z}} - 2\overline{\mathbf{z}}^T S\overline{\mathbf{z}} - \overline{\mathbf{u}}^T\overline{\mathbf{z}}$.
ii. $2\left(\overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}}\right) = \overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{u}}^T\overline{\mathbf{w}} + \overline{\mathbf{u}}^T\overline{\mathbf{z}}$.

We can now prove the analogue of Lemma 1.iii as follows:

$$2\left(\overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}}\right) = \overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{u}}^T\overline{\mathbf{w}} + \overline{\mathbf{u}}^T\overline{\mathbf{z}}$$

$$\Rightarrow 2\underbrace{\left(\overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}}\right)}_{\ge f(\overline{\mathbf{z}})-\delta} + \underbrace{\max(S\overline{\mathbf{w}}) - \overline{\mathbf{w}}^T S\overline{\mathbf{w}}}_{=f(\overline{\mathbf{w}})} = \max(S\overline{\mathbf{w}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{u}}^T\overline{\mathbf{w}} + \underbrace{\overline{\mathbf{u}}^T\overline{\mathbf{z}}}_{\le\delta}$$

$$\Rightarrow \quad \boxed{3\min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} \le 2f(\overline{\mathbf{z}}) + f(\overline{\mathbf{w}}) \le 3\delta + \max(S\overline{\mathbf{w}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{u}}^T\overline{\mathbf{w}}}$$

From this conclude that one of $(\overline{\mathbf{z}}, \overline{\mathbf{z}}), (\overline{\mathbf{w}}, \overline{\mathbf{w}})$ is a $(1/3 + \delta)$ −SNE of $\langle S, S^T\rangle$.

We now show that, for sufficiently small $\delta > 0$, if we find a $\delta-$KKT point $(\bar{s}, \overline{\mathbf{z}})$ whose Lagrange multiplier induces another $\delta-$KKT point $(\max(S\overline{\mathbf{w}}), \overline{\mathbf{w}})$, then we can construct a $\left(< \frac{1}{3}\right)$ −SNE point for $\langle S, S^T\rangle$.

Theorem 3. *For any* rational *matrix* $S \in [0,1]^{n\times n}$*, and any* $\frac{1}{27} > \delta > 0$*, if we are given a* δ*−KKT point* $(\max(S\overline{\mathbf{z}}), \overline{\mathbf{z}})$ *whose Lagrange multiplier* $(\overline{\mathbf{w}}, \overline{\mathbf{u}}, \bar{\zeta})$ *induces another* δ*−KKT point* $(\max(S\overline{\mathbf{w}}), \overline{\mathbf{w}})$*, then one of* $(\overline{\mathbf{z}}, \overline{\mathbf{z}}), (\overline{\mathbf{w}}, \overline{\mathbf{w}})$ *is a* $\left(< \frac{1}{3}\right)$*−SNE of* $\langle S, S^T\rangle$*.*

Proof. Assume for sake of contradiction that $\min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} \geq \frac{1}{3}$. We already proved in the previous theorem that

$$\frac{1}{3} \leq \min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} \leq \delta + \frac{\max(S\overline{\mathbf{w}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{u}}^T\overline{\mathbf{w}}}{3} \leq \delta + \frac{1}{3}$$

This implies that: $\boxed{\max(S\overline{\mathbf{w}}) - \overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{u}}^T\overline{\mathbf{w}} \geq 1 - 3\delta}$

Additionally,

$$\begin{aligned} &3\min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} \leq 2f(\overline{\mathbf{z}}) + f(\overline{\mathbf{w}}) - |f(\overline{\mathbf{z}}) - f(\overline{\mathbf{w}})| \leq 1 + 3\delta - |f(\overline{\mathbf{z}}) - f(\overline{\mathbf{w}})| \\ &\quad\Rightarrow |f(\overline{\mathbf{z}}) - f(\overline{\mathbf{w}})| \leq 1 + 3\delta - 3\min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} \leq 3\delta \end{aligned}$$

That is:

$$\boxed{\tfrac{1}{3} \leq \min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} \leq \min\{f(\overline{\mathbf{z}}), f(\overline{\mathbf{w}})\} + |f(\overline{\mathbf{z}}) - f(\overline{\mathbf{w}})| \leq \tfrac{1}{3} + 4\delta}$$

We now exploit the assumption that both $(\max(S\overline{\mathbf{z}}), \overline{\mathbf{z}})$ and $(\max(S\overline{\mathbf{w}}), \overline{\mathbf{w}})$ are δ−KKT points. We denote by $(\tilde{\mathbf{w}}, \tilde{\mathbf{u}}, \tilde{\zeta})$ the Lagrange multipliers corresponding to the latter point. We apply the stationarity condition for both of them:

$$\begin{aligned} \left.\begin{aligned} -S\overline{\mathbf{z}} - S^T\overline{\mathbf{z}} + S^T\overline{\mathbf{w}} = \overline{\mathbf{u}} + \mathbf{1}\bar{\zeta} \\ \bar{\zeta} = \overline{\mathbf{w}}^T S\overline{\mathbf{z}} - 2\overline{\mathbf{z}}^T S\overline{\mathbf{z}} - \overline{\mathbf{u}}^T\overline{\mathbf{z}} \end{aligned}\right\} &\Rightarrow \\ -\tilde{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\tilde{\mathbf{w}} + \overline{\mathbf{w}}^T S\tilde{\mathbf{w}} = \tilde{\mathbf{w}}^T\overline{\mathbf{u}} + \overline{\mathbf{w}}^T S\overline{\mathbf{z}} - 2\overline{\mathbf{z}}^T S\overline{\mathbf{z}} - \overline{\mathbf{u}}^T\overline{\mathbf{z}} &\Rightarrow \\ \boxed{\tilde{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} = \overline{\mathbf{w}}^T S\tilde{\mathbf{w}} - \overline{\mathbf{z}}^T S\tilde{\mathbf{w}} - \tilde{\mathbf{w}}^T\overline{\mathbf{u}} - (\overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}}) + \overline{\mathbf{u}}^T\overline{\mathbf{z}}} & \end{aligned}$$

Similarly:

$$\begin{aligned} \left.\begin{aligned} -S\overline{\mathbf{w}} - S^T\overline{\mathbf{w}} + S^T\tilde{\mathbf{w}} = \tilde{\mathbf{u}} + \mathbf{1}\tilde{\zeta} \\ \tilde{\zeta} = \tilde{\mathbf{w}}^T S\overline{\mathbf{w}} - 2\overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \tilde{\mathbf{u}}^T\overline{\mathbf{w}} \end{aligned}\right\} &\Rightarrow \\ -\overline{\mathbf{z}}^T S\overline{\mathbf{w}} - \overline{\mathbf{w}}^T S\overline{\mathbf{z}} + \tilde{\mathbf{w}}^T S\overline{\mathbf{z}} = \overline{\mathbf{z}}^T\tilde{\mathbf{u}} + \tilde{\mathbf{w}}^T S\overline{\mathbf{w}} - 2\overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \tilde{\mathbf{u}}^T\overline{\mathbf{w}} &\Rightarrow \\ \boxed{\tilde{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}} = \overline{\mathbf{z}}^T\tilde{\mathbf{u}} + \tilde{\mathbf{w}}^T S\overline{\mathbf{w}} - 2\overline{\mathbf{w}}^T S\overline{\mathbf{w}} - \tilde{\mathbf{u}}^T\overline{\mathbf{w}} + \overline{\mathbf{z}}^T S\overline{\mathbf{w}} + (\overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}})} & \end{aligned}$$

Combining the right-hand sides of these two equalities, we have:

$$\begin{aligned} 5\delta &\geq \underbrace{\overline{\mathbf{w}}^T S\tilde{\mathbf{w}}}_{\leq 3\delta} - \underbrace{\overline{\mathbf{z}}^T S\tilde{\mathbf{w}}}_{\geq 0} - \underbrace{\tilde{\mathbf{w}}^T\overline{\mathbf{u}}}_{\geq 0} + \underbrace{\overline{\mathbf{u}}^T\overline{\mathbf{z}}}_{\leq \delta} + \underbrace{\tilde{\mathbf{u}}^T\overline{\mathbf{w}}}_{\leq \delta} \\ &= \underbrace{\overline{\mathbf{z}}^T\tilde{\mathbf{u}}}_{\geq 0} + 2\underbrace{(\tilde{\mathbf{w}}^T S\overline{\mathbf{w}} - 2\overline{\mathbf{w}}^T S\overline{\mathbf{w}})}_{\geq f(\overline{\mathbf{w}})-\delta} - \underbrace{\tilde{\mathbf{w}}^T S\overline{\mathbf{w}}}_{\leq 1} + \underbrace{\overline{\mathbf{z}}^T S\overline{\mathbf{w}}}_{\geq 0} + 2\underbrace{(\overline{\mathbf{w}}^T S\overline{\mathbf{z}} - \overline{\mathbf{z}}^T S\overline{\mathbf{z}})}_{\geq f(\overline{\mathbf{z}})-\delta} \\ &\geq 2f(\overline{\mathbf{z}}) + 2f(\overline{\mathbf{w}}) - 4\delta - 1 \geq \frac{1}{3} - 4\delta \end{aligned}$$

which is impossible, for any $\delta < \frac{1}{27}$.

5 Extension to Asymmetric Games

In this section we shall demonstrate how we can exploit our approximation result for normalized symmetric bimatrix games, to the asymmetric case. In particular, we shall show how we can construct in polynomial time $\left(\frac{1}{3}+\delta\right)$ −NE points for any win lose bimatrix game, as well as for any normalized bimatrix game with maximum aggregate payoff at most 1 or at least $\frac{5}{3}$. To this direction, we propose a symmetrization of bimatrix games, which we shall prove to be approximation preserving for these games. The symmetrization is a straightforward generalization of the **BvN-symmetrization** of Brown and von Neumann for zero-sum games [6]. This symmetrization causes a relative blow-up in the action spaces of the players (as opposed to the well-known **GKT-symmetrization** of Gale, Kuhn and Tucker [16]), but will be shown to essentially preserve approximations of NE points, which is certainly not the case for the GKT-symmetrization. In particular, let $\langle R, C\rangle$ be any $m\times n$ bimatrix game. We construct a new, $mn\times mn$ matrix of payoffs S as follows:

$$\forall (i,j),(k,\ell)\in [m]\times[n],\ S_{ij,k\ell} = R_{i\ell}+C_{kj}$$

We then consider the symmetric game $\langle S, S^T\rangle$. We also consider the following mapping of strategies $\mathbf{z}\in\Delta_{mn}$ of $\langle S, S^T\rangle$ to a profile of "marginal strategies" $(\mathbf{x},\mathbf{y})$ of $\langle R, C\rangle$:

$$\forall (i,j)\in[m]\times[n],\ x_i=\sum_{\ell\in[n]} z_{i\ell}\ \wedge\ y_j=\sum_{k\in[m]} z_{kj}$$

Lemma 2. *Fix any (not necessarily normalized) bimatrix game $\langle R, C\rangle$, and the corresponding symmetric game $\langle S, S^T\rangle$ considered by the BvN-symmetrization. Then, any $(\overline{\mathbf{z}},\overline{\mathbf{z}})\in SNE(\varepsilon, S, S^T)$ indicates a profile $(\overline{\mathbf{x}},\overline{\mathbf{y}})\in NE(\varepsilon, R, C)$.*

Proof. First of all, observe that $\forall \mathbf{z}\in\Delta_{mn}$ and its BvN-image $(\mathbf{x},\mathbf{y})\in\Delta_m\times\Delta_n$,

$$\begin{aligned}
\mathbf{z}^T S\mathbf{z} &= \sum_{i\in[m]}\sum_{j\in[n]}\sum_{k\in[m]}\sum_{\ell\in[n]} z_{ij}z_{k\ell}\cdot(R_{i,\ell}+C_{k,j})\\
&=\sum_i\sum_\ell R_{i,\ell}\sum_k\sum_j z_{ij}z_{k\ell}+\sum_k\sum_j C_{k,j}\sum_i\sum_\ell z_{ij}z_{k\ell}\\
&=\sum_i\sum_\ell R_{i,\ell}\left(\sum_k z_{k\ell}\right)\left(\sum_j z_{ij}\right)+\sum_k\sum_j C_{k,j}\left(\sum_i z_{ij}\right)\left(\sum_\ell z_{k\ell}\right)\\
&=\mathbf{x}^T R\mathbf{y}+\mathbf{x}^T C\mathbf{y}
\end{aligned}$$

On the other hand, for approximate NE points of the symmetric game we have:

$$\begin{aligned}
& \overline{\mathbf{z}} \in NE(\varepsilon, S, S^T) \\
\Leftrightarrow \quad & \forall (i,j) \in [m]\times[n],\ \textstyle\sum_{k\in[m]}\sum_{\ell\in[n]}(R_{i,\ell}+C_{k,j})z_{k\ell} - \varepsilon \le \overline{\mathbf{z}}^T S \overline{\mathbf{z}} \\
\Leftrightarrow \quad & \forall (i,j) \in [m]\times[n],\ \textstyle\sum_{\ell\in[n]} R_{i,\ell} \underbrace{\sum_{k\in[m]} z_{k\ell}}_{=\bar{y}_\ell} + \sum_{k\in[m]} C_{k,j} \underbrace{\sum_{\ell\in[n]} z_{k\ell}}_{=\bar{x}_k} - \varepsilon \le \overline{\mathbf{z}}^T S \overline{\mathbf{z}} \\
\Leftrightarrow \quad & \forall (i,j) \in [m]\times[n],\ R_{i,\star}\overline{\mathbf{y}} + {C_{\star,j}}^T \overline{\mathbf{x}} - \varepsilon \le \overline{\mathbf{x}}^T R \overline{\mathbf{y}} + \overline{\mathbf{x}}^T C \overline{\mathbf{y}} \\
\Leftrightarrow \quad & \boxed{\forall (\mathbf{x},\mathbf{y}) \in \Delta_m \times \Delta_n,\ \mathbf{x}^T R \overline{\mathbf{y}} + \overline{\mathbf{x}}^T C \mathbf{y} - \varepsilon \le \overline{\mathbf{x}}^T R \overline{\mathbf{y}} + \overline{\mathbf{x}}^T C \overline{\mathbf{y}}}
\end{aligned}$$

By trying all $(\mathbf{x}, \overline{\mathbf{y}})$ and $(\overline{\mathbf{x}}, \mathbf{y})$, it is trivial to observe that $(\overline{\mathbf{x}}, \overline{\mathbf{y}}) \in NE(\varepsilon, R, C)$.

The following theorem is a direct consequence of the above approximation-preserving symmetrization, and our approximation algorithm for normalized symmetric bimatrix games.

Theorem 4. *There is a polynomial-time algorithm that constructs* $\left(\frac{1}{3}+\delta\right)-NE$ *points for (i) win-lose bimatrix games, and (ii) games* $\langle R, C\rangle$ *with* $\max(R+C) \le 1$ *or* $\max(R+C) \ge \frac{5}{3}$.

Proof. For the win-lose case, we may safely exclude games having in the payoff bimatrix a $(1,1)-$element, since this would be a trivial pure NE of the game. Similarly, if a normalized game $\langle R, C\rangle$ has maximum aggregate payoff $\max(R+C) \ge \frac{5}{3}$, then obviously the pure strategy profile attaining it is a $\frac{1}{3}-$NE point of the game. Therefore, it suffices to prove the claim only for the class of normalized games with aggregate payoff at most 1. Since $\max(R+C) \le 1$, we can be sure that the payoff matrix S in the BvN-symmetrization is also normalized. Therefore, we can apply our algorithm for the symmetric case to construct a point $(\overline{\mathbf{z}}, \overline{\mathbf{z}}) \in NE\left(\frac{1}{3}+\delta, S, S^T\right)$, for any fixed $\delta > 0$. We have already proved in the previous lemma that the marginal profile is equally good for the original game: $(\overline{\mathbf{x}}, \overline{\mathbf{y}}) \in NE\left(\frac{1}{3}+\delta, R, C\right)$.

6 Presentation of Random Games Generator and BIMATRIX-NASH Relaxations

Our main platform for the experimental analysis that we conducted was Matlab (2007b, The MathWorks, Natick, MA). We have also used the CVX modeling system for disciplined convex programs [18], particularly for the description (and solution) of some of the relaxations of (SMS), as well as for the description of the CE polytope and the optimization of linear functions in it. Our main goal in this experimental study is to focus on the *quality* of the provided solutions, *given their polynomial-time solvability* which is assured either by the (SMS) instance being recognized as convex QP, or by being approximated by the proposed polynomial-time solvable relaxations, or finally by optimizing linear functions in the CE-polytope of the game. Therefore, we have conducted extensive experimental tests looking for worst-case instances for each of the considered relaxations of the game.

We start our discussion by explaining the random game generator. Consequently we present each of the (pure) relaxation techniques that we have implemented. We end this experimental part with the presentation of our experimental results both on the pure relaxation techniques and on hybrid methods that combine subsets of relaxations and return the best approximation per game. All our experiments were conducted on random symmetric win-lose bimatrix games. The focus on symmetric games was for the sake of simplicity. The choice of win-lose rather than normalized games was due to our observation that harder cases for our relaxations appear by the former type of games, at least on the sequences of games produced by our random game generator.

6.1 Pseudo-random Game Generator

Our pseudo-random game generator creates $m \times n$ bimatrix games $\langle R, C\rangle$, in which each payoff matrix is generated by the `randn(m,n)` routine of Matlab. This routine returns an $m \times n$ matrix containing pseudo-random values drawn from the standard normal distribution with mean 0 and standard deviation 1. The sequence of pseudo-random numbers produced by `randn` is determined by the *internal state* (seed) of the uniform pseudo-random number generator of Matlab. It uses $m \cdot n$ uniform values from that default stream to generate each normal value. In order to create independent instances in different runs of the experiment, we reset the default stream (by renewing randomly the state) at the beginning of each new execution (as a function of the global clock value at runtime). Of course, it is also possible to enforce the repetition of exactly the same fixed state (and consequently, the same stream of uniform values) that would allow various experimentations to be run on the same (pseudo-random) set of games. Nevertheless, in this phase we have chosen to create independent runs, since our main objective is to investigate hard cases for the various approximation methods.

Our game generator supports the option to normalize or not the produced game. Normalization means re-scaling of the payoffs so that for each player the min payoff is exactly 0 and the max payoff is exactly 1.

We can also create win-lose bimatrix games, by discretizing the payoff values either to 0 (if non-positive) or to 1 (if positive). We may even use a different discretization threshold, in order to opt for either sparse or dense games, with respect to ones in the payoff matrices. Finally, our generator supports (optionally) the avoidance of win-lose games which can be trivially handled and/or be simplified. In particular, we have the option to avoid (i) “all-ones” rows in R, since such a row weakly dominates any other row for the row player and therefore implies the existence of a pure NE point in the game; (ii) “all-zeros” rows in R, since such a row is again weakly dominated and cannot disturb an approximate NE point found in the residual game. For similar reasons, we may choose to avoid (iii) “all-ones” columns and (iv) “all-zeros” columns in C. We also avoid the $(1, 1)-$elements in the bimatrix, which is also a trivial pure NE point. We try to make all these simplifications without affecting significantly the randomness of the created game. To this direction, we start by changing (uniformly

and independently) each (1,1)-element to either a $(0,1)-$ or to a $(1,0)-$element. Consequently, we switch one random element of an "all-ones" row in R to 0. Similarly, we switch the first random element of an "all-zeros" row in R to 1, provided that this change does not create a new $(1,1)-$element. We handle in a similar fashion the columns of C.

6.2 KKT Relaxation

Our first approach, which is in accordance with the theoretical part of this paper, is to actually return an arbitrary KKT point of (SMS) as an approximate NE point of a symmetric bimatrix game $\langle R, R^T \rangle$. For this phase we have used the `quadprog` function of Matlab, that attempts to converge to (or even reach) a local solution, unless the problem is strictly convex. Indeed, we first check the current instance of (SMS) for convexity, and in the convex case we solve it using the SeDuMi solver via the CVX environment. In the non-convex case, we call the `quadprog` function. In case that it either reaches, or converges to a local optimum, we return it as an approximate NE point of the game. Otherwise, we report failure of the method and we do not count the particular game in our quest for worst-case instances. This is because the previously reported theoretical results on (KKTSMS) are valid only at KKT points of (SMS). Nevertheless, we keep in mind the number of (non-convex) games that were not solved by this method.

6.3 RLT Relaxation

The Reformulation-Linearization-Technique (RLT) is a method that generates tight LP relaxations of either discrete combinatorial optimization or continuous (nonconvex) polynomial programming problems. For the particular case of mixed 0-1 linear programs (MILPs) in n binary variables, the RLT generates an $n-$level hierarchy which at the $n-$th level provides an explicit algebraic characterization of the convex hull of feasible solutions.

The method essentially consists of two steps, the reformulation step in which additional valid inequalities are generated, and the linearization step in which each nonlinear monomial in either the objective function or in any of the constraints, is replaced by a new (continuous) variable. For continuous (non-convex) quadratic optimization programs with linear constraints, the reformulation-step is as follows: For each variable x_j that has bound-constraints $\ell_j \leq x_j \leq u_j$, the non-negative expressions $(x_j - \ell_j)$ and $(u_j - x_j)$ are called **bound-factors**. For any linear constraint $\mathbf{a}^T\mathbf{x} \geq b$ (other than the bound constraints) the expression $(\mathbf{a}^T\mathbf{x} - b)$ is called a **constraint-factor**. The new constraints are produced by the reformulation step by requiring the non-negativity of *all* products of subsets (say, pairs) of (either bound- or constraint-) factors. Consequently, the linearization step substitutes any monomial (eg, of degree 2) by new variables. The same substitution also applies in the possibly non-linear objective function of the program. The resulting program is indeed an LP relaxation of the original (continuous) polynomial program. For more details, the reader is deferred to [30, Chapters 7-8]. In this work we have applied the RLT to the SMS,

by considering all the possible pairs of bound and constraint factors. In order to achieve as tight an approximation as possible, we have added to the SMS the constraints $\{1 \geq \alpha \geq 0;\ \forall j \in [n], x_j \leq 1\}$, exploiting the fact that the games that we consider are normalized. This set of constraints may be redundant for SMS itself, but it also contributes to the RLT relaxation with quite useful constraints. The result of this technique is the following RLT-relaxation (the variables $\beta \in \mathbb{R},\ \gamma \in \mathbb{R}^n,\ W \in \mathbb{R}^{n \times n}$ are added during the linearization phase):

$$
\text{(RLTSMS)}\quad
\begin{array}{rl}
\multicolumn{2}{c}{\textbf{minimize} \quad \alpha - \sum_{i\in[n]}\sum_{j\in[n]} R_{i,j}W_{i,j}} \\
\text{s.t. } \beta - \sum_j (R_{i,j}+R_{k,j})\gamma_j + \sum_j\sum_\ell R_{i,j}R_{k,\ell}W_{j,\ell} \geq & 0,\ i,k\in[n] \\
\alpha - \sum_j R_{i,j}x_j - \gamma_k + \sum_j R_{i,j}W_{j,k} \geq & 0,\ i,k\in[n] \\
-x_i - x_j + W_{i,j} \geq & -1,\ i,j\in[n] \\
\gamma_k - \sum_j R_{i,j}\gamma_j \geq & 0,\ i,k\in[n] \\
x_i - \sum_j W_{i,j} = & 0,\ i\in[n] \\
\sum_j x_j = & 1, \\
-x_i - \alpha + \gamma_i \geq & -1,\ i\in[n] \\
x_i - \gamma_i \geq & 0,\ i\in[n] \\
\beta - \sum_j R_{i,j}\gamma_j \geq & 0,\ i\in[n] \\
\sum_i \gamma_i - \alpha = & 0, \\
\alpha - \beta \geq & 0, \\
\multicolumn{2}{c}{\beta \geq 0; \quad \gamma_i \geq 0,\ i\in[n]; \quad W_{i,j}\geq 0,\ i,j\in[n]}
\end{array}
$$

Observe that some constraints of (SMS) are missing from (RLTSMS). This is because it is well known (and trivial to show) that all the constraints of (SMS) are implied by the constraints of (RLTSMS) [30, Proposition 8.1]. We have also excluded some of the RLT constraints that were clearly implied by the ones presented in (RLTSMS). We could possibly have avoided more constraints, that are also induced by the constraints in (RLTSMS). Nevertheless, our primary goal is to experimentally test the approximability of NE points via the RLT relaxation, given its polynomiality, and not (at least, not yet) to optimize the computational time. In order to solve (RLTSMS), we have used the CVX modeling system for disciplined convex programming [18].

6.4 Doubly Positive Semidefinite Programming Relaxation

In this section we exploit the fact that (SMS) could be transformed so that the objective be linear, by using the substitution $W = \mathbf{x}\cdot\mathbf{x}^T$. Of course, this is a non-linear equality constraint. In order to tackle the apparent intractability of this formulation as well, we relax the additional equality constraint to the non-linear convex constraint: $W - \mathbf{x}\cdot\mathbf{x}^T \geq \mathbf{O}$, where the matrix comparisons are component-wise. This is equivalent to demanding that the matrix

$$
Z = \begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix}
$$

be positive semidefinite (PSD). Apart from the PSD constraint $Z \succcurlyeq 0$, there is also an obvious non-negativity constraint: $Z \geq \mathbf{O}$. This kind of semidefinite relaxations are called doubly-positive semidefinite (DPSDP) relaxations (eg, [7]). Their advantage is that linear optimization over the cone of DPSDP matrices can be performed using techniques for self-dual cones, such as SeDuMi [31]. So, we have implemented the following DPSDP-relaxation of (SMS), which we then formulate in the CVX environment and have it solved by the SeDuMi solver:

$$
\text{(DPSD)}\quad
\boxed{\begin{array}{ll}
\textbf{minimize} & \alpha - \sum_i \sum_j R_{i,j} W_{i,j} \\
\text{s.t } \alpha - \sum_j R_{i,j} x_j \geq 0, & i \in [n] \\
\sum_j x_j = 1, & \\
W_{i,j} - W_{j,i} = 0, & i,j \in [n] \\
Z \equiv \begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix} \succcurlyeq 0, & \\
Z \leq \mathbf{E}, & \\
Z \geq \mathbf{O} &
\end{array}}
$$

6.5 Using Projections of Extreme CE Points

Our final approach for producing approximate NE points is based on an idea that we presented in [21]. In that work we proved (among other things) that in the following parameterized non-linear program the set of optimal solutions completely (and exclusively) describes the Nash equilibrium set of the normalized bimatrix game $\langle R, C \rangle$, for any value $t \in (0,1)$:

$$
\text{(KS(t))}\quad
\boxed{\begin{array}{ll}
\textbf{minimize} & \sum_i \sum_j [tR_{i,j} + (1-t)C_{i,j}] W_{i,j} - \mathbf{1}^T W^T [tR + (1-t)C] W^T \mathbf{1} \\
\text{s.t.} & \forall i,k \in [m],\ \sum_{j\in[n]} (R_{i,j} - R_{k,j}) W_{i,j} \geq 0 \\
& \forall j,\ell \in [n],\ \sum_{i\in[m]} (C_{i,j} - C_{i,\ell}) W_{i,j} \geq 0 \\
& \sum_{i\in[m]} \sum_{j\in[n]} W_{i,j} = 1 \\
\forall (i,j) \in [m]\times[n], & W_{i,j} \geq 0
\end{array}}
$$

In particular, it was shown in [21] that the marginal distributions $\hat{\mathbf{x}} = \hat{W}\mathbf{1}$, $\hat{\mathbf{y}} = \hat{W}^T\mathbf{1}$ of any optimal solution $\hat{W}$ of (KS(t)) comprise a NE point of $\langle R, C \rangle$, and vice versa. Since in the present work we deal with polynomial-time approximations of NE points in symmetric bimatrix games, in our last attempt indeed we tried to optimize various linear functions in the feasible space of (KS(t)). The most prominent linear objective was to minimize the objective $(R \circ C) \bullet W = \mathbf{1}^T (R \circ C \circ W) \mathbf{1}$. That is, we return as approximate NE points the marginals of an optimal solution to the following linear program (which we again modeled in CVX):

$$
\text{(BMXCEV4)}\quad
\boxed{\begin{array}{ll}
\textbf{minimize} & \sum_i \sum_j [R_{i,j} \cdot R_{j,i}] W_{i,j} \\
\text{s.t.} & \forall i,k \in [m],\ \sum_{j\in[n]} (R_{i,j} - R_{k,j}) W_{i,j} \geq 0 \\
& \sum_{i\in[m]} \sum_{j\in[n]} W_{i,j} = 1 \\
\forall (i,j) \in [m]\times[n], & W_{i,j} \geq 0
\end{array}}
$$

7 Experimental Results

In this section we summarize the results of our experimental analysis on the implemented relaxation methods for (SMS).

7.1 Pure Relaxations

First we conducted a series of experiments on each of the pure relaxations that we have rum $500K$ random instances of 10×10 games (without touching the random games produced by the generator). For each game we accepted the produced approximate NE point only if the corresponding solver managed to converge to a solution. Otherwise we excluded the game from the worst-case accounting and only kept a log of unsolved games. The results of these experiments are summarized in Table 1.

Consequently we ran the same experiments, but this time we demanded from the random game generator to avoid the appearance of some trivial cases (cf. Section 6.1). Our experimental results in this case are presented in Table 2.

We have also experimented on the distribution of games solved by each game, as a function of the epsilon-value. To this direction, we have focused on 10×10 games, and we have run $10K$ random (untouched) instances per pure relaxation method. Our findings are summarized by the graphs presented in Figure 1.

It is worth mentioning that by means of approximation the best pure method is (KKTSMS). It is also noted that avoiding trivial cases in the produced random games (cf. Subsection 6.1) has a significant impact mainly on the approximations of (KKTSMS) and (BMXCEV4). Concerning the number of games that are solved, the winner is (BMXCEV4) with the other three methods being comparable.

Table 1. Experimental results for worst-case approximation among 500K random 10×10 symmetric win-lose games

	RLTSMS	KKTSMS	DPSDP	BMXCEV4
worst-case epsilon	0.51432	0.22222	0.6	0.49836
# unsolved games	112999	110070	0	405
worst-case round	10950	15484	16690	12139

Table 2. Experimental results for worst-case approximation among 500K random 10×10 symmetric win-lose games which avoid $(1,1)-$elements, $(1,*)-$ and $(0,*)-$rows, $(*,1)-$ and $(*,0)-$columns in the payoff matrix

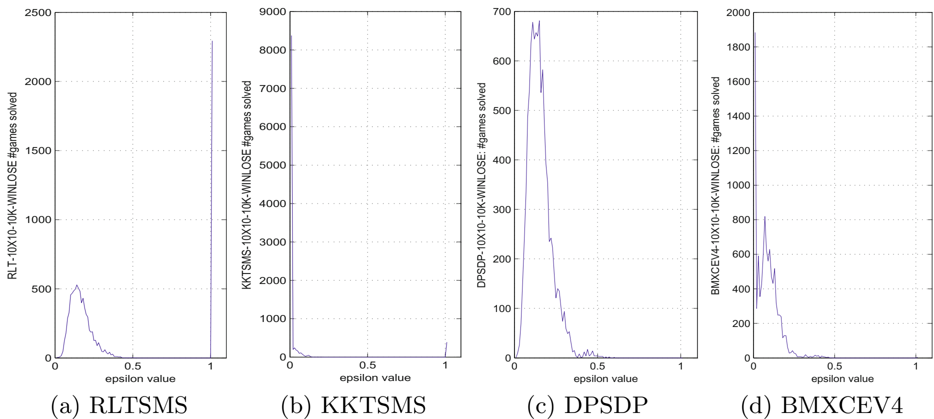
	RLTSMS	KKTSMS	DPSDP	BMXCEV4
worst-case epsilon	0.41835	0.08333	0.51313	0.21203
# unsolved games	11183	32195	0	1553
worst-case round	45062	42043	55555	17923

Table 3. Experimental results for worst-case approximations of hybrid methods among 500K random 10×10 symmetric win-lose games

	KKTSMS+BMXCEV4	KKTSMS+RLT+BMXCEV4	KKTSMS+RLT+DPSDP
worst-case epsilon	0.47881	0.47881	0.54999
# unsolved games	0	0	0
worst-case round	652	1776	7737

Table 4. Experimental results for worst-case approximations of hybrid methods among 500K random 10×10 symmetric win-lose games which avoid $(1,1)-$elements, $(1,*)-$ and $(0,*)-$rows, $(*,1)-$ and $(*,0)-$columns in the payoff matrix

	KKTSMS+BMXCEV4	KKTSMS+RLT+BMXCEV4	KKTSMS+RLT+DPSDP
worst-case epsilon	0.0.08576	0.088576	0.28847
# unsolved games	0	0	0
worst-case round	157185	397418	186519



(a) RLTSMS (b) KKTSMS (c) DPSDP (d) BMXCEV4

Fig. 1. Distribution of games solved for particular values of approximation, in runs of $10K$ random 10×10 symmetric win-lose games. The games that remain unsolved in each case accumulate at the epsilon value 1.

7.2 Hybrid Relaxations

We have finally conducted experiments that use a "best-of" approach to produce approximate NE points in randomly produced symmetric bimatrix games. The goal of these hybrid approaches is two-fold: To explore whether a combination of methods decreases the approximation ratio even below the ratio of (KKTSMS) which seems to be the best pure method. But also to minimize the number of unsolved games, which is significant even for (BMXCEV4)

which is the winner under this objective. We have run experiments on the hybrid methods KKTSMS+BMXCEV4, KKTSMS+RLT+BMXCEV4 and KKTSMS+RLT+DPSDP. The reason for choosing KKTSMS as a common component in all these combinations is the significantly smaller execution time of this method, and also its extremely good behavior in most of the instances that are solvable by it (cf. Figure 1).

The experimental results of these hybrid approaches are summarized in Table 3 (for raw random games) and Table 4 (for random games that avoid trivial situations).

From these experiments it is clear that hybrid methods do not significantly help (at least in worst case) the approximation ratio of the provided solutions, but on the other hand we observe that there are no unsolved instances anymore, since the different approaches do not have ill-conditioned instances in common. It is also observed that it is actually the KKTSMS and the BMXCEV4 methods that act complementarily and also assure the best observed behavior. Our runs on the hybrid KKTSMS+RLT+BMXCEV4 shows that the RLT method does not really contribute to the quality of the solutions. It is also observed that the hybrid KKTSMS+RLT+DPSDP has rather poor approximation guarantees, which implies that some ill-conditioned games for KKTSMS are also hard cases for the other two methods.

8 Conclusions

In this work we have presented a simple quadratic formulation for the problem of computing (exact) NE points of symmetric bimatrix games. We then showed how to construct approximate NE points, with approximation ratio arbitrarily close to $\frac{1}{3}$, in polynomial time. We also observed that indeed there exist even better approximate NE points, which would be polynomial-time constructible given *any* initial (exact) KKT point with a Lagrange dual that is also a KKT point. Indeed, we strongly suspect that there is a polynomial-time construction even in the case where this demand for a primal-dual pair of KKT points is not satisfied. Nevertheless, we were not able to formally prove this until now, and it remains an open question for future investigation. We also showed that our approach also works for any win lose game, or for any asymmetric game with maximum aggregate payoff either at most 1, or at least $\frac{5}{3}$. We are currently investigating our techniques directly to the general asymmetric case.

Our experimental analysis of various (pure and hybrid) approximation methods indicates that it is most likely that we can do better than the theoretically proved bound of $\frac{1}{3}$, which almost holds also for the asymmetric case. Of course, this remains to be proved.

Acknowledgements. The authors wish to thank Christos Papadimitriou for bringing to their attention the BvN-symmetrization method.

References

1. Addario-Berry, L., Olver, N., Vetta, A.: A polynomial time algorithm for finding nash equilibria in planar win-lose games. Journal of Graph Algorithms and Applications 11(1), 309–319 (2007)
2. Adsul, B., Garg, J., Mehta, R., Sohoni, M.: Rank-1 bimatrix games: A homeomorphism and a polynomial time algorithm. In: Proc. of 43rd ACM Symp. on Th. of Comp., STOC 2011 (2011)
3. Althöfer, I.: On sparse approximations to randomized strategies and convex combinations. Linear Algebra and Applications 199, 339–355 (1994)
4. Bertsekas, D.: Nonlinear Programming, 2nd edn. Athena Scientific, Belmont (2003)
5. Bosse, H., Byrka, J., Markakis, E.: New algorithms for approximate nash equilibria in bimatrix games. In: Deng, X., Graham, F.C. (eds.) WINE 2007. LNCS, vol. 4858, pp. 17–29. Springer, Heidelberg (2007)
6. Brown, G.W., von Neumann, J.: Solutions of games by differential equations. Annals of Mathematical Studies 24, 73–79 (1950)
7. Burer, S.: On the copositive representation of binary and continuous nonconvex quadratic programs. Mathematical Programming 120, 479–495 (2009)
8. Chen, X., Deng, X.: Settling the complexity of 2-player nash equilibrium. In: Proc. of 47th IEEE Symp. on Found. of Comp. Sci. (FOCS 2006), pp. 261–272. IEEE Comp. Soc. Press, Los Alamitos (2006)
9. Chen, X., Deng, X., Teng, S.H.: Computing nash equilibria: Approximation and smoothed complexity. In: Proc. of 47th IEEE Symp. on Found. of Comp. Sci. (FOCS 2006), pp. 603–612. IEEE Comp. Soc. Press, Los Alamitos (2006)
10. Codenotti, B., Leoncini, M., Resta, G.: Efficient computation of nash equilibria for very sparse win-lose bimatrix games. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 232–243. Springer, Heidelberg (2006)
11. Conitzer, V., Sandholm, T.: Complexity results about nash equilibria. In: Proc. of 18th Int. Joint Conf. on Art. Intel. (IJCAI 2003), pp. 765–771. Morgan Kaufmann, San Francisco (2003)
12. Daskalakis, C., Goldberg, P.W., Papadimitriou, C.H.: The complexity of computing a nash equilibrium. SIAM Journal on Computing 39(1), 195–259 (2009)
13. Daskalakis, C., Mehta, A., Papadimitriou, C.: A note on approximate nash equilibria. In: Spirakis, P.G., Mavronicolas, M., Kontogiannis, S.C. (eds.) WINE 2006. LNCS, vol. 4286, pp. 297–306. Springer, Heidelberg (2006)
14. Daskalakis, C., Mehta, A., Papadimitriou, C.H.: Progress in approximate nash equilibrium. In: Proc. of 8th ACM Conf. on El. Comm. (EC 2007), pp. 355–358 (2007)
15. Daskalakis, C., Papadimitriou, C.H.: Three player games are hard. Technical Report TR05-139, Electr. Coll. on Comp. Compl., ECCC (2005)
16. Gale, D., Kuhn, H.W., Tucker, A.W.: On symmetric games. Contributions to Game Theory 1, 81–87 (1950)
17. Gilboa, I., Zemel, E.: Nash and correlated equilibria: Some complexity considerations. Games & Econ. Behavior 1, 80–93 (1989)
18. Grant, M., Boyd, S.: CVX: Matlab software for disciplined convex programming, version 1.21 (February 2011), http://cvxr.com/cvx
19. Kannan, R., Theobald, T.: Games of fixed rank: A hierarchy of bimatrix games. Economic Theory 42, 157–173 (2010)

20. Kontogiannis, S., Panagopoulou, P., Spirakis, P.: Polynomial algorithms for approximating nash equilibria of bimatrix games. In: Spirakis, P.G., Mavronicolas, M., Kontogiannis, S.C. (eds.) WINE 2006. LNCS, vol. 4286, pp. 286–296. Springer, Heidelberg (2006)
21. Kontogiannis, S., Spirakis, P.: Exploiting concavity in bimatrix games: New polynomially tractable subclasses. In: Serna, M., Shaltiel, R., Jansen, K., Rolim, J. (eds.) APPROX 2010. LNCS, vol. 6302. Springer, Heidelberg (2010), http://www.cs.uoi.gr/~kontog/pubs/approx10paper-full.pdf
22. Kontogiannis, S., Spirakis, P.: Well supported approximate equilibria in bimatrix games. ALGORITHMICA 57, 653–667 (2010)
23. Lee, G.M., Tam, N.N., Yen, N.D.: Quadratic Programming and Affine Variational Inequalities – A Qualitative Study. Nonconvex Optimization and its Applications. Springer, Heidelberg (2005)
24. Lemke, C.E., Howson Jr., J.T.: Equilibrium points of bimatrix games. Journal of the Society for Industrial and Applied Mathematics 12, 413–423 (1964)
25. Lipton, R., Markakis, E., Mehta, A.: Playing large games using simple strategies. In: Proc. of 4th ACM Conf. on El. Comm (EC 2003), pp. 36–41. ACM, New York (2003)
26. Mangasarian, O.L., Stone, H.: Two-person nonzero-sum games and quadratic programming. Journal of Mathematical Analysis and Applications 9(3), 348–355 (1964)
27. Morgenstern, O., von Neumann, J.: The Theory of Games and Economic Behavior. Princeton University Press, Princeton (1947)
28. Moulin, H., Vial, J.P.: Strategically zero-sum games: The class of games whose completely mixed equilibria cannot be improved upon. Int. Journal of Game Theory 7(3/4), 201–221 (1978)
29. Savani, R., von Stengel, B.: Exponentially many steps for finding a nash equilibrium in a bimatrix game. In: Proc. of 45th IEEE Symp. on Found. of Comp. Sci. (FOCS 2004), pp. 258–267 (2004)
30. Sherali, H.D., Adams, W.P.: A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems. Kluwer Academic Publishers, Dordrecht (1998)
31. Sturm, J.F.: Using SeDuMi 1.02, a matlab toolbox for optimization over symmetric cones. Optimization Methods and Software 11-12, 625–653 (1999)
32. Tsaknakis, H., Spirakis, P.: An optimization approach for approximate nash equilibria. In: Deng, X., Graham, F.C. (eds.) WINE 2007. LNCS, vol. 4858, pp. 42–56. Springer, Heidelberg (2007)
33. Tsaknakis, H., Spirakis, P.: A graph spectral approach for computing approximate nash equilibria. In: Saberi, A. (ed.) WINE 2010. LNCS, vol. 6484, pp. 378–390. Springer, Heidelberg (2010)
34. Ye, Y.: On the complexity of approximating a kkt point of quadratic programming. Mathematical Programming 80, 195–211 (1998)

Metaheuristic Optimization: Algorithm Analysis and Open Problems

Xin-She Yang

Mathematics and Scientific Computing, National Physical Laboratory,
Teddington, Middlesex TW11 0LW, UK

Abstract. Metaheuristic algorithms are becoming an important part of modern optimization. A wide range of metaheuristic algorithms have emerged over the last two decades, and many metaheuristics such as particle swarm optimization are becoming increasingly popular. Despite their popularity, mathematical analysis of these algorithms lacks behind. Convergence analysis still remains unsolved for the majority of metaheuristic algorithms, while efficiency analysis is equally challenging. In this paper, we intend to provide an overview of convergence and efficiency studies of metaheuristics, and try to provide a framework for analyzing metaheuristics in terms of convergence and efficiency. This can form a basis for analyzing other algorithms. We also outline some open questions as further research topics.

1 Introduction

Optimization is an important subject with many important application, and algorithms for optimization are diverse with a wide range of successful applications [10,11]. Among these optimization algorithms, modern metaheuristics are becoming increasingly popular, leading to a new branch of optimization, called metaheuristic optimization. Most metaheuristic algorithms are nature-inspired [8,29,32], from simulated annealing [20] to ant colony optimization [8], and from particle swarm optimization [17] to cuckoo search [35]. Since the appearance of swarm intelligence algorithms such as PSO in the 1990s, more than a dozen new metaheuristic algorithms have been developed and these algorithms have been applied to almost all areas of optimization, design, scheduling and planning, data mining, machine intelligence, and many others. Thousands of research papers and dozens of books have been published [8,9,11,19,29,32,33].

Despite the rapid development of metaheuristics, their mathematical analysis remains partly unsolved, and many open problems need urgent attention. This difficulty is largely due to the fact the interaction of various components in metaheuristic algorithms are highly nonlinear, complex, and stochastic. Studies have attempted to carry out convergence analysis [1,22], and some important results concerning PSO were obtained [7]. However, for other metaheuristics such as firefly algorithms and ant colony optimization, it remains an active, challenging topic. On the other hand, even we have not proved or cannot prove their convergence, we still can compare the performance of various algorithms.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 21–32, 2011.
© Springer-Verlag Berlin Heidelberg 2011

This has indeed formed a majority of current research in algorithm development in the research community of optimization and machine intelligence [9,29,33].

In combinatorial optimization, many important developments exist on complexity analysis, run time and convergence analysis [25,22]. For continuous optimization, no-free-lunch-theorems do not hold [1,2]. As a relatively young field, many open problems still remain in the field of randomized search heuristics [1]. In practice, most assume that metaheuristic algorithms tend to be less complex for implementation, and in many cases, problem sizes are not directly linked with the algorithm complexity. However, metaheuristics can often solve very tough NP-hard optimization, while our understanding of the efficiency and convergence of metaheuristics lacks far behind.

Apart from the complex interactions among multiple search agents (making the mathematical analysis intractable), another important issue is the various randomization techniques used for modern metaheuristics, from simple randomization such as uniform distribution to random walks, and to more elaborate Lévy flights [5,24,33]. There is no unified approach to analyze these mathematically. In this paper, we intend to review the convergence of two metaheuristic algorithms including simulated annealing and PSO, followed by the new convergence analysis of the firefly algorithm. Then, we try to formulate a framework for algorithm analysis in terms of Markov chain Monte Carlo. We also try to analyze the mathematical and statistical foundations for randomization techniques from simple random walks to Lévy flights. Finally, we will discuss some of important open questions as further research topics.

2 Convergence Analysis of Metaheuristics

The formulation and numerical studies of various metaheuristics have been the main focus of most research studies. Many successful applications have demonstrated the efficiency of metaheuristics in various context, either through comparison with other algorithms and/or applications to well-known problems. In contrast, the mathematical analysis lacks behind, and convergence analysis has been carried out for only a minority few algorithms such as simulated annealing and particle swarm optimization [7,22]. The main approach is often for very simplified systems using dynamical theory and other ad hoc approaches. Here in this section, we first review the simulated annealing and its convergence, and we move onto the population-based algorithms such as PSO. We then take the recently developed firefly algorithm as a further example to carry out its convergence analysis.

2.1 Simulated Annealing

Simulated annealing (SA) is one of the widely used metaheuristics, and is also one of the most studies in terms of convergence analysis [4,20]. The essence of simulated annealing is a trajectory-based random walk of a single agent, starting from an initial guess $\boldsymbol{x}_0$. The next move only depends on the current state or location and the acceptance probability p. This is essentially a Markov chain whose transition probability from the current state to the next state is given by

$$p = \exp\Big[-\frac{\Delta E}{k_B T}\Big], \tag{1}$$

where k_B is Boltzmann's constant, and T is the temperature. Here the energy change ΔE can be linked with the change of objective values. A few studies on the convergence of simulated annealing have paved the way for analysis for all simulated annealing-based algorithms [4,15,27]. Bertsimas and Tsitsiklis provided an excellent review of the convergence of SA under various assumptions [4,15]. By using the assumptions that SA forms an inhomogeneous Markov chain with finite states, they proved a probabilistic convergence function P, rather than almost sure convergence, that

$$\max P\Big[\boldsymbol{x}_i(t) \in S_* \Big| \boldsymbol{x}_0\Big] \geq \frac{A}{t^{\alpha}}, \tag{2}$$

where S_* is the optimal set, and A and α are positive constants [4]. This is for the cooling schedule $T(t) = d/\ln(t)$, where t is the iteration counter or pseudo time. These studies largely used Markov chains as the main tool. We will come back later to a more general framework of Markov chain Monte Carlo (MCMC) in this paper [12,14].

2.2 PSO and Convergence

Particle swarm optimization (PSO) was developed by Kennedy and Eberhart in 1995 [17,18], based on the swarm behaviour such as fish and bird schooling in nature. Since then, PSO has generated much wider interests, and forms an exciting, ever-expanding research subject, called swarm intelligence. PSO has been applied to almost every area in optimization, computational intelligence, and design/scheduling applications.

The movement of a swarming particle consists of two major components: a stochastic component and a deterministic component. Each particle is attracted toward the position of the current global best $\boldsymbol{g}^*$ and its own best location $\boldsymbol{x}_i^*$ in history, while at the same time it has a tendency to move randomly.

Let $\boldsymbol{x}_i$ and $\boldsymbol{v}_i$ be the position vector and velocity for particle i, respectively. The new velocity and location updating formulas are determined by

$$\boldsymbol{v}_i^{t+1} = \boldsymbol{v}_i^t + \alpha \boldsymbol{\epsilon}_1 [\boldsymbol{g}^* - \boldsymbol{x}_i^t] + \beta \boldsymbol{\epsilon}_2 [\boldsymbol{x}_i^* - \boldsymbol{x}_i^t]. \tag{3}$$

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^{t+1}, \tag{4}$$

where $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_2$ are two random vectors, and each entry taking the values between 0 and 1. The parameters α and β are the learning parameters or acceleration constants, which can typically be taken as, say, $\alpha \approx \beta \approx 2$.

There are at least two dozen PSO variants which extend the standard PSO algorithm, and the most noticeable improvement is probably to use inertia function $\theta(t)$ so that $\boldsymbol{v}_i^t$ is replaced by $\theta(t)\boldsymbol{v}_i^t$ where $\theta \in [0,1]$ [6]. This is equivalent to introducing a virtual mass to stabilize the motion of the particles, and thus the algorithm is expected to converge more quickly.

The first convergence analysis of PSO was carried out by Clerc and Kennedy in 2002 [7] using the theory of dynamical systems. Mathematically, if we ignore the random factors, we can view the system formed by (3) and (4) as a dynamical system. If we focus on a single particle i and imagine that there is only one particle in this system, then the global best $\boldsymbol{g}^*$ is the same as its current best $\boldsymbol{x}_i^*$. In this case, we have

$$\boldsymbol{v}_i^{t+1} = \boldsymbol{v}_i^t + \gamma(\boldsymbol{g}^* - \boldsymbol{x}_i^t), \quad \gamma = \alpha + \beta, \tag{5}$$

and

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^{t+1}. \tag{6}$$

Considering the 1D dynamical system for particle swarm optimization, we can replace $\boldsymbol{g}^*$ by a parameter constant p so that we can see if or not the particle of interest will converge towards p. By setting $u_t = p - x(t+1)$ and using the notations for dynamical systems, we have a simple dynamical system

$$v_{t+1} = v_t + \gamma u_t, \quad u_{t+1} = -v_t + (1-\gamma)u_t, \tag{7}$$

or

$$Y_{t+1} = AY_t, \quad A = \begin{pmatrix} 1 & \gamma \\ -1 & 1-\gamma \end{pmatrix}, \quad Y_t = \begin{pmatrix} v_t \\ u_t \end{pmatrix}. \tag{8}$$

The general solution of this dynamical system can be written as $Y_t = Y_0 \exp[At]$. The system behaviour can be characterized by the eigenvalues λ of A, and we have $\lambda_{1,2} = 1 - \gamma/2 \pm \sqrt{\gamma^2 - 4\gamma}/2$. It can be seen clearly that $\gamma = 4$ leads to a bifurcation. Following a straightforward analysis of this dynamical system, we can have three cases. For $0 < \gamma < 4$, cyclic and/or quasi-cyclic trajectories exist. In this case, when randomness is gradually reduced, some convergence can be observed. For $\gamma > 4$, non-cyclic behaviour can be expected and the distance from Y_t to the center $(0,0)$ is monotonically increasing with t. In a special case $\gamma = 4$, some convergence behaviour can be observed. For detailed analysis, please refer to Clerc and Kennedy [7]. Since p is linked with the global best, as the iterations continue, it can be expected that all particles will aggregate towards the the global best.

2.3 Firefly Algorithm, Convergence and Chaos

Firefly Algorithm (FA) was developed by Yang [32,34], which was based on the flashing patterns and behaviour of fireflies. In essence, each firefly will be attracted to brighter ones, while at the same time, it explores and searches for prey randomly. In addition, the brightness of a firefly is determined by the landscape of the objective function.

The movement of a firefly i is attracted to another more attractive (brighter) firefly j is determined by

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \beta_0 e^{-\gamma r_{ij}^2}(\boldsymbol{x}_j^t - \boldsymbol{x}_i^t) + \alpha\, \boldsymbol{\epsilon}_i^t, \tag{9}$$

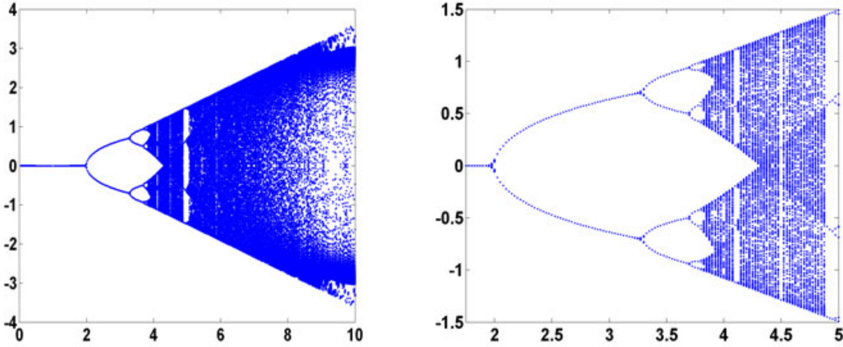


Fig. 1. The chaotic map of the iteration formula (13) in the firefly algorithm and the transition between from periodic/multiple states to chaos

where the second term is due to the attraction. The third term is randomization with α being the randomization parameter, and $\boldsymbol{\epsilon}_i^t$ is a vector of random numbers drawn from a Gaussian distribution or other distributions. Obviously, for a given firefly, there are often many more attractive fireflies, then we can either go through all of them via a loop or use the most attractive one. For multiple modal problems, using a loop while moving toward each brighter one is usually more effective, though this will lead to a slight increase of algorithm complexity.

Here is $\beta_0 \in [0, 1]$ is the attractiveness at $r = 0$, and $r_{ij} = ||\boldsymbol{x}_i - \boldsymbol{x}_j||_2$ is the ℓ_2-norm or Cartesian distance. For other problems such as scheduling, any measure that can effectively characterize the quantities of interest in the optimization problem can be used as the 'distance' r.

For most implementations, we can take $\beta_0 = 1$, $\alpha = O(1)$ and $\gamma = O(1)$. It is worth pointing out that (9) is essentially a random walk biased towards the brighter fireflies. If $\beta_0 = 0$, it becomes a simple random walk. Furthermore, the randomization term can easily be extended to other distributions such as Lévy flights [16,24].

We now can carry out the convergence analysis for the firefly algorithm in a framework similar to Clerc and Kennedy's dynamical analysis. For simplicity, we start from the equation for firefly motion without the randomness term

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \beta_0 e^{-\gamma r_{ij}^2} (\boldsymbol{x}_j^t - \boldsymbol{x}_i^t). \tag{10}$$

If we focus on a single agent, we can replace $\boldsymbol{x}_j^t$ by the global best g found so far, and we have

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \beta_0 e^{-\gamma r_i^2} (g - \boldsymbol{x}_i^t), \tag{11}$$

where the distance r_i can be given by the ℓ_2-norm $r_i^2 = ||g - \boldsymbol{x}_i^t||_2^2$. In an even simpler 1-D case, we can set $y_t = g - \boldsymbol{x}_i^t$, and we have

$$y_{t+1} = y_t - \beta_0 e^{-\gamma y_t^2} y_t. \tag{12}$$

We can see that γ is a scaling parameter which only affects the scales/size of the firefly movement. In fact, we can let $u_t = \sqrt{\gamma} y_t$ and we have

$$u_{t+1} = u_t[1 - \beta_0 e^{-u_t^2}]. \tag{13}$$

These equations can be analyzed easily using the same methodology for studying the well-known logistic map

$$u_{t+1} = \lambda u_t (1 - u_t). \tag{14}$$

The chaotic map is shown in Fig. 1, and the focus on the transition from periodic multiple states to chaotic behaviour is shown in the same figure.

As we can see from Fig. 1 that convergence can be achieved for $\beta_0 < 2$. There is a transition from periodic to chaos at $\beta_0 \approx 4$. This may be surprising, as the aim of designing a metaheuristic algorithm is to try to find the optimal solution efficiently and accurately. However, chaotic behaviour is not necessarily a nuisance; in fact, we can use it to the advantage of the firefly algorithm. Simple chaotic characteristics from (14) can often be used as an efficient mixing technique for generating diverse solutions. Statistically, the logistic mapping (14) with $\lambda = 4$ for the initial states in (0,1) corresponds a beta distribution

$$B(u, p, q) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} u^{p-1} (1-u)^{q-1}, \tag{15}$$

when $p = q = 1/2$. Here $\Gamma(z)$ is the Gamma function

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt. \tag{16}$$

In the case when $z = n$ is an integer, we have $\Gamma(n) = (n-1)!$. In addition, $\Gamma(1/2) = \sqrt{\pi}$. From the algorithm implementation point of view, we can use higher attractiveness β_0 during the early stage of iterations so that the fireflies can explore, even chaotically, the search space more effectively. As the search continues and convergence approaches, we can reduce the attractiveness β_0 gradually, which may increase the overall efficiency of the algorithm. Obviously, more studies are highly needed to confirm this.

2.4 Markov Chain Monte Carlo

From the above convergence analysis, we know that there is no mathematical framework in general to provide insights into the working mechanisms, the stability and convergence of a give algorithm. Despite the increasing popularity of metaheuristics, mathematical analysis remains fragmental, and many open problems need urgent attention.

Monte Carlo methods have been applied in many applications [28], including almost all areas of sciences and engineering. For example, Monte Carlo methods are widely used in uncertainty and sensitivity analysis [21]. From the statistical point of view, most metaheuristic algorithms can be viewed in the framework of

Markov chains [14,28]. For example, simulated annealing [20] is a Markov chain, as the next state or new solution in SA only depends on the current state/solution and the transition probability. For a given Markov chain with certain ergodicity, a stability probability distribution and convergence can be achieved.

Now if look at the PSO closely using the framework of Markov chain Monte Carlo [12,13,14], each particle in PSO essentially forms a Markov chain, though this Markov chain is biased towards to the current best, as the transition probability often leads to the acceptance of the move towards the current global best. Other population-based algorithms can also be viewed in this framework. In essence, all metaheuristic algorithms with piecewise, interacting paths can be analyzed in the general framework of Markov chain Monte Carlo. The main challenge is to realize this and to use the appropriate Markov chain theory to study metaheuristic algorithms. More fruitful studies will surely emerge in the future.

3 Search Efficiency and Randomization

Metaheuristics can be considered as an efficient way to produce acceptable solutions by trial and error to a complex problem in a reasonably practical time. The complexity of the problem of interest makes it impossible to search every possible solution or combination, the aim is to find good feasible solutions in an acceptable timescale. There is no guarantee that the best solutions can be found, and we even do not know whether an algorithm will work and why if it does work. The idea is to have an efficient but practical algorithm that will work most the time and is able to produce good quality solutions. Among the found quality solutions, it is expected some of them are nearly optimal, though there is no guarantee for such optimality.

The main components of any metaheuristic algorithms are: intensification and diversification, or exploitation and exploration [5,33]. Diversification means to generate diverse solutions so as to explore the search space on the global scale, while intensification means to focus on the search in a local region by exploiting the information that a current good solution is found in this region. This is in combination with with the selection of the best solutions.

As discussed earlier, an important component in swarm intelligence and modern metaheuristics is randomization, which enables an algorithm to have the ability to jump out of any local optimum so as to search globally. Randomization can also be used for local search around the current best if steps are limited to a local region. Fine-tuning the randomness and balance of local search and global search is crucially important in controlling the performance of any metaheuristic algorithm.

Randomization techniques can be a very simple method using uniform distributions, or more complex methods as those used in Monte Carlo simulations [28]. They can also be more elaborate, from Brownian random walks to Lévy flights.

3.1 Gaussian Random Walks

A random walk is a random process which consists of taking a series of consecutive random steps. Mathematically speaking, let u_N denotes the sum of each consecutive random step s_i, then u_N forms a random walk

$$u_N = \sum_{i=1}^{N} s_i = s_1 + ... + s_N = u_{N-1} + s_N, \tag{17}$$

where s_i is a random step drawn from a random distribution. This suggests that the next state u_N will only depend the current existing state u_{N-1} and the motion or transition u_N from the existing state to the next state. In theory, as the number of steps N increases, the central limit theorem implies that the random walk (17) should approaches a Gaussian distribution. In addition, there is no reason why each step length should be fixed. In fact, the step size can also vary according to a known distribution. If the step length obeys the Gaussian distribution, the random walk becomes the standard Brownian motion [16,33].

From metaheuristic point of view, all paths of search agents form a random walk, including a particle's trajectory in simulated annealing, a zig-zag path of a particle in PSO, or the piecewise path of a firefly in FA. The only difference is that transition probabilities are different, and change with time and locations.

Under simplest assumptions, we know that a Gaussian distribution is stable. For a particle starts with an initial location $\boldsymbol{x}_0$, its final location $\boldsymbol{x}_N$ after N time steps is

$$\boldsymbol{x}_N = \boldsymbol{x}_0 + \sum_{i=1}^{N} \alpha_i s_i, \tag{18}$$

where $\alpha_i > 0$ is a parameters controlling the step sizes or scalings. If s_i is drawn from a normal distribution $N(\mu_i, \sigma_i^2)$, then the conditions of stable distributions lead to a combined Gaussian distribution

$$\boldsymbol{x}_N \sim N(\mu_*, \sigma_*^2), \quad \mu_* = \sum_{i=1}^{N} \alpha_i \mu_i, \quad \sigma_*^2 = \sum_{i=1}^{N} \alpha_i [\sigma_i^2 + (\mu_* - \mu_i)^2]. \tag{19}$$

We can see that that the mean location changes with N and the variances increases as N increases, this makes it possible to reach any areas in the search space if N is large enough.

A diffusion process can be viewed as a series of Brownian motion, and the motion obeys the Gaussian distribution. For this reason, standard diffusion is often referred to as the Gaussian diffusion. As the mean of particle locations is obviously zero if $\mu_i = 0$, their variance will increase linearly with t. In general, in the d-dimensional space, the variance of Brownian random walks can be written as

$$\sigma^2(t) = |v_0|^2 t^2 + (2dD)t, \tag{20}$$

where v_0 is the drift velocity of the system. Here $D = s^2/(2\tau)$ is the effective diffusion coefficient which is related to the step length s over a short time interval

τ during each jump. If the motion at each step is not Gaussian, then the diffusion is called non-Gaussian diffusion. If the step length obeys other distribution, we have to deal with more generalized random walks. A very special case is when the step length obeys the Lévy distribution, such a random walk is called Lévy flight or Lévy walk.

3.2 Randomization via Lévy Flights

In nature, animals search for food in a random or quasi-random manner. In general, the foraging path of an animal is effectively a random walk because the next move is based on the current location/state and the transition probability to the next location. Which direction it chooses depends implicitly on a probability which can be modelled mathematically [3,24]. For example, various studies have shown that the flight behaviour of many animals and insects has demonstrated the typical characteristics of Lévy flights [24,26]. Subsequently, such behaviour has been applied to optimization and optimal search, and preliminary results show its promising capability [30,24].

In general, Lévy distribution is stable, and can be defined in terms of a characteristic function or the following Fourier transform

$$F(k) = \exp[-\alpha |k|^{\beta}], \quad 0 < \beta \leq 2, \tag{21}$$

where α is a scale parameter. The inverse of this integral is not easy, as it does not have nay analytical form, except for a few special cases [16,23]. For the case of $\beta = 2$, we have $F(k) = \exp[-\alpha k^2]$, whose inverse Fourier transform corresponds to a Gaussian distribution. Another special case is $\beta = 1$, which corresponds to a Cauchy distribution

For the general case, the inverse integral

$$L(s) = \frac{1}{\pi} \int_0^{\infty} \cos(ks) \exp[-\alpha |k|^{\beta}] dk, \tag{22}$$

can be estimated only when s is large. We have

$$L(s) \to \frac{\alpha \, \beta \, \Gamma(\beta) \sin(\pi \beta / 2)}{\pi |s|^{1+\beta}}, \quad s \to \infty. \tag{23}$$

Lévy flights are more efficient than Brownian random walks in exploring unknown, large-scale search space. There are many reasons to explain this efficiency, and one of them is due to the fact that the variance of Lévy flights takes the following form

$$\sigma^2(t) \sim t^{3-\beta}, \quad 1 \leq \beta \leq 2, \tag{24}$$

which increases much faster than the linear relationship (i.e., $\sigma^2(t) \sim t$) of Brownian random walks.

Studies show that Lévy flights can maximize the efficiency of resource searches in uncertain environments. In fact, Lévy flights have been observed among foraging patterns of albatrosses and fruit flies [24,26,30]. In addition, Lévy flights have many applications. Many physical phenomena such as the diffusion of fluorenscent molecules, cooling behavior and noise could show Lévy-flight characteristics under the right conditions [26].

4 Open Problems

It is no exaggeration to say that metahueristic algorithms have been a great success in solving various tough optimization problems. Despite this huge success, there are many important questions which remain unanswered. We know how these heuristic algorithms work, and we also partly understand why these algorithms work. However, it is difficult to analyze mathematically why these algorithms are so successful, though significant progress has been made in the last few years [1,22]. However, many open problems still remain.

For all population-based metaheuristics, multiple search agents form multiple interacting Markov chains. At the moment, theoretical development in these areas are still at early stage. Therefore, the mathematical analysis concerning of the rate of convergence is very difficult, if not impossible. Apart from the mathematical analysis on a limited few metaheuristics, convergence of all other algorithms has not been proved mathematically, at least up to now. Any mathematical analysis will thus provide important insight into these algorithms. It will also be valuable for providing new directions for further important modifications on these algorithms or even pointing out innovative ways of developing new algorithms.

For almost all metaheuristics including future new algorithms, an important issue to be addresses is to provide a balanced trade-off between local intensification and global diversification [5]. At present, different algorithm uses different techniques and mechanism with various parameters to control this, they are far from optimal. Important questions are: Is there any optimal way to achieve this balance? If yes, how? If not, what is the best we can achieve?

Furthermore, it is still only partly understood why different components of heuristics and metaheuristics interact in a coherent and balanced way so that they produce efficient algorithms which converge under the given conditions. For example, why does a balanced combination of randomization and a deterministic component lead to a much more efficient algorithm (than a purely deterministic and/or a purely random algorithm)? How to measure or test if a balance is reached? How to prove that the use of memory can significantly increase the search efficiency of an algorithm? Under what conditions?

In addition, from the well-known No-Free-Lunch theorems [31], we know that they have been proved for single objective optimization for finite search domains, but they do not hold for continuous infinite domains [1,2]. In addition, they remain unproved for multiobjective optimization. If they are proved to be true (or not) for multiobjective optimization, what are the implications for algorithm development? Another important question is about the performance comparison. At the moment, there is no agreed measure for comparing performance of different algorithms, though the absolute objective value and the number of function evaluations are two widely used measures. However, a formal theoretical analysis is yet to be developed.

Nature provides almost unlimited ways for problem-solving. If we can observe carefully, we are surely inspired to develop more powerful and efficient new generation algorithms. Intelligence is a product of biological evolution in nature.

Ultimately some intelligent algorithms (or systems) may appear in the future, so that they can evolve and optimally adapt to solve NP-hard optimization problems efficiently and intelligently.

Finally, a current trend is to use simplified metaheuristic algorithms to deal with complex optimization problems. Possibly, there is a need to develop more complex metaheuristic algorithms which can truly mimic the exact working mechanism of some natural or biological systems, leading to more powerful next-generation, self-regulating, self-evolving, and truly intelligent metaheuristics.

References

1. Auger, A., Doerr, B.: Theory of Randomized Search Heuristics: Foundations and Recent Developments. World Scientific, Singapore (2010)
2. Auger, A., Teytaud, O.: Continuous lunches are free plus the design of optimal optimization algorithms. Algorithmica 57(1), 121–146 (2010)
3. Bell, W.J.: Searching Behaviour: The Behavioural Ecology of Finding Resources. Chapman & Hall, London (1991)
4. Bertsimas, D., Tsitsiklis, J.: Simulated annealing. Stat. Science 8, 10–15 (1993)
5. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: Overview and conceptural comparison. ACM Comput. Surv. 35, 268–308 (2003)
6. Chatterjee, A., Siarry, P.: Nonlinear inertia variation for dynamic adapation in particle swarm optimization. Comp. Oper. Research 33, 859–871 (2006)
7. Clerc, M., Kennedy, J.: The particle swarm - explosion, stability, and convergence in a multidimensional complex space. IEEE Trans. Evolutionary Computation 6, 58–73 (2002)
8. Dorigo, M.: Optimization, Learning and Natural Algorithms. PhD thesis. Politecnico di Milano, Italy (1992)
9. Engelbrecht, A.P.: Fundamentals of Computational Swarm Intelligence. Wiley, Chichester (2005)
10. Floudas, C.A., Pardalos, P.M.: A Collection of Test Problems for Constrained Global Optimization Algorithms. LNCS, vol. 455. Springer, Heidelberg (1990)
11. Floudas, C.A., Pardolos, P.M.: Encyclopedia of Optimization, 2nd edn. Springer, Heidelberg (2009)
12. Gamerman, D.: Markov Chain Monte Carlo. Chapman & Hall/CRC (1997)
13. Geyer, C.J.: Practical Markov Chain Monte Carlo. Statistical Science 7, 473–511 (1992)
14. Ghate, A., Smith, R.: Adaptive search with stochastic acceptance probabilities for global optimization. Operations Research Lett. 36, 285–290 (2008)
15. Granville, V., Krivanek, M., Rasson, J.P.: Simulated annealing: A proof of convergence. IEEE Trans. Pattern Anal. Mach. Intel. 16, 652–656 (1994)
16. Gutowski, M.: Lévy flights as an underlying mechanism for global optimization algorithms. ArXiv Mathematical Physics e-Prints (June 2001)
17. Kennedy, J., Eberhart, R.C.: Particle swarm optimization. In: Proc. of IEEE International Conference on Neural Networks, Piscataway, NJ, pp. 1942–1948 (1995)
18. Kennedy, J., Eberhart, R.C.: Swarm intelligence. Academic Press, London (2001)
19. Holland, J.: Adaptation in Natural and Artificial systems. University of Michigan Press, Ann Anbor (1975)
20. Kirkpatrick, S., Gellat, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220, 670–680 (1983)

21. Matthews, C., Wright, L., Yang, X.S.: Sensitivity Analysis, Optimization, and Sampling Methodds Applied to Continous Models. National Physical Laboratory Report, UK (2009)
22. Neumann, F., Witt, C.: Bioinspired Computation in Combinatorial Optimization: Algorithms and Their Computational Complexity. Springer, Heidelberg (2010)
23. Nolan, J.P.: Stable distributions: models for heavy-tailed data. American University (2009)
24. Pavlyukevich, I.: Lévy flights, non-local search and simulated annealing. J. Computational Physics 226, 1830–1844 (2007)
25. Rebennack, S., Arulselvan, A., Elefteriadou, L., Pardalos, P.M.: Complexity analysis for maximum flow problems with arc reversals. J. Combin. Optimization 19(2), 200–216 (2010)
26. Reynolds, A.M., Rhodes, C.J.: The Lévy fligth paradigm: random search patterns and mechanisms. Ecology 90, 877–887 (2009)
27. Steinhöfel, K., Albrecht, A.A., Wong, C.-K.: Convergence analysis of simulated annealing-based algorithms solving flow shop scheduling problems. In: Bongiovanni, G., Petreschi, R., Gambosi, G. (eds.) CIAC 2000. LNCS, vol. 1767, pp. 277–290. Springer, Heidelberg (2000)
28. Sobol, I.M.: A Primer for the Monte Carlo Method. CRC Press, Boca Raton (1994)
29. Talbi, E.G.: Metaheuristics: From Design to Implementation. Wiley, Chichester (2009)
30. Viswanathan, G.M., Buldyrev, S.V., Havlin, S., da Luz, M.G.E., Raposo, E.P., Stanley, H.E.: Lévy flight search patterns of wandering albatrosses. Nature 381, 413–415 (1996)
31. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimisation. IEEE Transaction on Evolutionary Computation 1, 67–82 (1997)
32. Yang, X.S.: Nature-Inspired Metaheuristic Algorithms. Luniver Press (2008)
33. Yang, X.S.: Engineering Optimization: An Introduction with Metaheuristic Applications. John Wiley & Sons, Chichester
34. Yang, X.S.: Firefly algorithm, stochastic test functions and design optimisation. Int. J. Bio-Inspired Computation 2, 78–84 (2010)
35. Yang, X.S., Deb, S.: Engineering optimization by cuckoo search. Int. J. Math. Modelling & Num. Optimization 1, 330–343 (2010)

Convexity and Optimization of Condense Discrete Functions

Emre Tokgöz[1], Sara Nourazari[2], and Hillel Kumin[3]

[1,3] School of Industrial Engineering, University of Oklahoma, Norman, OK, 73019, U.S.A.
[1] Department of Mathematics, University of Oklahoma, Norman, OK, 73019, U.S.A.
[2] George Mason University, Department of Systems Engineering and Operations Research, Fairfax, Virginia, 22030, U.S.A.
{Emre.Tokgoz-1,hkumin}@ou.edu, snouraza@masonlive.gmu.edu

Abstract. A function with one integer variable is defined to be integer convex by Fox [3] and Denardo [1] if its second forward differences are positive. In this paper, condense discrete convexity of nonlinear discrete multivariable functions with their corresponding Hessian matrices is introduced which is a generalization of the integer convexity definition of Fox [3] and Denardo [1] to higher dimensional space $\mathbb{Z}^n$. In addition, optimization results are proven for C^1 condense discrete convex functions assuming that the given condense discrete convex function is C^1. Yüceer [17] proves convexity results for a certain class of discrete convex functions and shows that the restriction of the adaptation of Rosenbrook's function from real variables to discrete variables does not yield a discretely convex function. Here it is shown that the adaptation of Rosenbrook's function considered in [17] is a condense discrete convex function where the set of local minimums is also the the set of global minimums.

Keywords: Integer programming, mathematical programming, discrete convex function, real convex function.

1 Introduction

In real convex analysis the convexity of a C^2 function can be obtained by checking whether or not the corresponding Hessian matrix is positive definite. This result has important applications to optimization problems for real variable C^2 functions. In particular, a C^2 function $f : \mathbb{R}^n \to \mathbb{R}$ is strictly convex if and only if the corresponding Hessian matrix is positive definite, and, therefore any local minimum of f is also the global minimum and vice versa. A Hessian matrix in real convex analysis also identifies the closed form convexity conditions for the corresponding C^2 function in local settings (see [15] for details).

In discrete convex analysis, Denardo [1], Fox [3], and many others in the literature define a single variable discrete function to be convex if the first forward differences of the given function are increasing or at least nondecreasing. A multivariable discrete L-convex function is defined to be the generalization of the

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 33–42, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Lovász extension of submodular set functions in [12] by Murota. $L^{\#}-$convex functions are defined in [4] by Fujishige and Murota. The concept of M-convex functions is introduced by Murota in [11] and that of $M^{\#}$ convex functions by Murota-Shioura in [14]. The discrete analogue of Hessian matrices corresponding to multivariable discrete $L, L^{\#}, M,$ and $M^{\#}$ functions are introduced by Hirai and Murota [5], and Moriguchi and Murota [10]. Important applications of $L, L^{\#}, M,$ and $M^{\#}$ discrete convex/concave functions appear in network flow problems (see [13] for details). The convexity properties of nonlinear integer variable, integer valued objective functions are investigated by Favati and Tardella [2] where algorithmic approaches are also presented. Kiselman ([6], [7] and [8]) defines the convex envelope, canonical extension and lateral convexity of multivariable discrete functions where the second difference of a function $f : \mathbb{Z}^n \to \mathbb{R}$ is introduced to define lateral convexity.

Let S be a subspace of a discrete n-dimensional space. A function $f : S \to R$ is defined to be discrete convex by Yüceer [17] (using Miller's [9] definition) if for all $x, y \in S$ and $\alpha > 0$ we have

$$\alpha f(x) + (1-\alpha) f(y) \geq \min_{u \in N(z)} f(u)$$

where $z = \alpha x + (1-\alpha)y$, $N(z) = \{u \in S : \|u - z\| < 1\}$, and $\|u\| = \max\{|u_i| : 1 \leq i \leq n\}$. This discrete convex function definition yields nonnegative second forward differences in each component, and a symmetric matrix of second forward cross differences. By imposing additional submodularity conditions on discrete convex functions, the concept of strong discrete convexity is introduced in [17]. A strong discrete convex function has a corresponding positive semidefinite matrix of second forward differences which has practical and computational implications. D-convex and semistrictly quasi D-convex functions are introduced in [16] by Ui where D-convex functions have a unified form that includes discretely convex, integrally convex, M convex, $M^{\natural}$ convex, L convex, and $L^{\natural}$ convex functions in local settings.

In this paper, we introduce a definition of condense discrete convexity of a nonlinear real extensible closed form function $\beta : \mathbb{Z}^n \to \mathbb{R}$, which is a generalization of the integer convexity definition of Fox [3] and Denardo [1] for a one variable discrete function to nonlinear multivariable discrete variable functions. A discrete Hessian matrix H consisting of second differences $\nabla_{ij}\beta$ $(1 \leq i, j \leq n)$ corresponding to a condense discrete convex function $\beta : \mathbb{Z}^n \to \mathbb{R}$ in local settings is introduced, and convexity results are obtained for condense discrete functions similar to the convexity results obtained in real convex analysis. The discrete Hessian matrix H is shown to be symmetric, linear, and vanishes when the condense discrete function is affine.

Yüceer [17] states that the restriction of any continuous function to a discrete space does not necessarily yield a discrete convex function where an adaptation of Rosenbrook's function is illustrated as an example. In this paper, we show that the discretization of the Rosenbrook's function from continuous variables to integer variables is a condense discrete convex function.

To obtain minimization results for a given condense discrete convex function, we require the given condense discrete convex function to be C^1. After defining the local and global minimum of condense discrete convex functions, we obtain convexity results for C^1 condense discrete convex functions.

2 Convexity and Optimization of Condense Discrete Functions

In this section, we first introduce the first and second differences of an n-integer variable function. Second, we introduce the condense discrete convexity of an n-integer variable function which is a generalization of the integer convexity definition of Denardo [1] and Fox [3] for one variable discrete functions. The second partial derivative of a C^2 real convex function f in $\mathbb{R}^n$ becomes the second difference of a condense discrete convex function in $\mathbb{Z}^n$ when we change the domain of f from $\mathbb{R}^n$ to $\mathbb{Z}^n$ which is the main idea of the condense discrete convexity concept. Therefore the condense discrete convexity definition is similar to the real convexity definition where we check whether the given discrete variable function is C^2 or not. This has practical applications when we consider C^2 discrete functions such as discrete variable polynomials.

2.1 Convexity of Condense Discrete Functions

Similar to the difference operator definition of Kiselman [7], we define the first difference of an integer variable function $f : \mathbb{Z}^n \to \mathbb{R}$ by

$$\nabla_i f(x) = f(x + e_i) - f(x),$$

and the difference of the first difference, namely the second difference of f is defined by

$$\nabla_{ij}(f(x)) = f(x + e_i + e_j) - f(x + e_i) - f(x + e_j) + f(x),$$

where e_i represents the integer vectors of unit length at the ith position of the function f. We define a condense discrete convex set D to be the set of points that coincides with a real convex set on the integer lattice which is large enough to support the second difference of a given condense discrete function. We assume that the union of condense discrete convex sets are discrete convex sets as well. The following definition of an n-integer variable function holds for a certain class of discrete functions. The definition of a condense discrete convex function is based on its quadratic approximation in a condense discrete convex neighborhood $D \subset \mathbb{Z}^n$.

Definition 1. A discrete function $f : D \to \mathbb{R}$ on a condense discrete convex set $D \subset \mathbb{Z}^n$ is defined to be condense discrete convex if its quadratic approximation $\frac{1}{2}x^T Ax$ in the neighborhood D is strictly positive where A is the symmetric coefficient matrix of the quadratic approximation of f. f is called condense discrete

concave if $-f$ is condense discrete convex. A is called the discrete coefficient matrix of f.

Proposition 1. Let $f : D \to \mathbb{R}$ be defined on a condense discrete convex set $D \subset \mathbb{Z}^n$ with its quadratic approximation $\frac{1}{2}x^T Ax$. The coefficient matrix A corresponding to f is the symmetric matrix $\left[\nabla_{ij}(f)\right]_{n\times n}$.

Proof. We first prove the symmetry of the matrix $\left[\nabla_{ij}(f)\right]_{n\times n}$.

$$\begin{aligned}\nabla_{ij} f(x) &= \nabla_i \left(f(x+e_j) - f(x)\right)\\ &= f(x+e_i+e_j) - f(x+e_i) - f(x+e_j) + f(x)\\ &= \nabla_j \left(f(x+e_i) - f(x)\right)\\ &= \nabla_j \left(\nabla_i f(x)\right) = \nabla_{ji} f(x).\end{aligned}$$

Assuming that A is symmetric, for all i and j,

$$\begin{aligned}\nabla_{ij}(f(x)) &= \frac{1}{2}\nabla_i \left((x+e_j)^T A(x+e_j) - x^T Ax\right)\\ &= \frac{1}{2}\nabla_i \left(x^T A(x+e_j) + e_j^T A(x+e_j) - x^T Ax\right)\\ &= \frac{1}{2}\nabla_i \left(x^T Ax + x^T Ae_j + e_j^T Ax + e_j^T Ae_j - x^T Ax\right)\\ &= \frac{1}{2}\nabla_i \left(x^T Ae_j + e_j^T Ax + e_j^T Ae_j\right)\\ &= \frac{1}{2}\left((x+e_i)^T Ae_j - x^T Ae_j + e_j^T A(x+e_i) - e_j^T Ax\right)\\ &= \frac{1}{2}\left(x^T Ae_j + e_i^T Ae_j - x^T Ae_j + e_j^T Ax + e_j^T Ae_i - e_j^T Ax\right)\\ &= \frac{1}{2}\left(e_i^T Ae_j + e_j^T Ae_i\right)\\ &= a_{ij}.\end{aligned}$$

Therefore

$$A_f = \left[a_{ij}\right]_{n\times n} = \left[\nabla_{ij} f\right]_{n\times n}. \qquad \square$$

Proposition 2. The coefficient matrix A_f of $f : D \to \mathbb{R}$ given above in proposition 1 satisfies the properties of the discrete Hessian matrix corresponding to real convex functions. That is, A_f is linear with respect to the condense discrete functions, symmetric, and vanishes when f is discrete affine.

Proof. Let $f_1 : S_1 \to \mathbb{R}$ and $f_2 : S_2 \to \mathbb{R}$ be condense discrete functions with the corresponding coefficient matrices A_{f_1} and A_{f_2}, respectively. Then

$$\begin{aligned}\left[\nabla_{ij}(f_1+f_2)\right]_{n\times n} &= A_{f_1+f_2}\\ &= A_{f_1} + A_{f_2}\\ &= \left[\nabla_{ij}(f_1)\right]_{n\times n} + \left[\nabla_{ij}(f_2)\right]_{n\times n}\end{aligned}$$

which also proves the linearity of the second difference operator with respect to the condense discrete functions. The symmetry condition is proven in proposition 1.

Considering the condense discrete affine function f,

$$f(x) = \sum_{i=1}^{n} b_i x_i$$

the second difference operator vanishes since $\nabla_i(f) = b_i$ and $\nabla_{ij}(f) = 0$ for all i and j. □

Theorem 1. A function $f : D \to \mathbb{R}$ is condense discrete convex if and only if the corresponding discrete Hessian matrix is positive definite in D.

Proof. Consider the discrete function

$$f(x) = x^T A_f x = \sum_{i,j=1}^{2} a_{ij} x_i x_j$$

where $a_{ij} \in \mathbb{R}$ for all $1 \leq i, j \leq 2$, and $x \in \mathbb{Z}^2$. We prove the case for 2×2 matrix and $n \times n$ matrix case follows similarly. Suppose A_f is positive definite.

Case 1. If we let $x = (1, 0)$, then

$$f(x) = a_{11}x_1^2 + 2a_{12}x_1x_2 + a_{22}x_2^2 = a_{11} > 0.$$

Case 2. If we let $x = (0, 1)$, then

$$f(x) = a_{11}x_1^2 + 2a_{12}x_1x_2 + a_{22}x_2^2 = a_{22} > 0.$$

To show $A_f > 0$ for any $x \neq 0$ consider the following cases.

Case 1. If we let $x = (x_1, 0)$ with $x_1 \neq 0$. Then,

$$f(x) = a_{11}x_1^2 + 2a_{12}x_1x_2 + a_{22}x_2^2 = a_{11}x_1^2 > 0 \Leftrightarrow a_{11} > 0.$$

Case 2. If we let $x = (x_1, x_2)$ with $x_2 \neq 0$. Let $x_1 = tx_2$ for some $t \in \mathbb{R}$. Therefore we have

$$f(x) = \left(a_{11}t^2 + 2a_{12}t + a_{22}\right) x_2^2$$

where $f(x) > 0 \Leftrightarrow \varphi(t) = a_{11}t^2 + 2a_{12}t + a_{22} > 0$ since $x_2 \neq 0$. Note that

$$\begin{aligned} \varphi'(t) &= 2a_{11}t + 2a_{12} = 0 \\ &\Rightarrow t^* = -\frac{a_{12}}{a_{11}} \\ \varphi''(t) &= 2a_{11}. \end{aligned}$$

If $a_{11} > 0$ then

$$\begin{aligned} \varphi(t) \geq \varphi(t^*) = \varphi\left(-\frac{a_{12}}{a_{11}}\right) &= \frac{-a_{12}^2}{a_{11}} + a_{22} \\ &= \frac{1}{a_{11}} \det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}. \end{aligned}$$

Therefore if $a_{11} > 0$ and the determinant given above is positive then $\varphi(t) > 0$ for all $t \in \mathbb{R}$. Conversely, if $f(x) > 0$ for every $x \neq 0$ then $\varphi(t) > 0$ for some t, therefore

$$\varphi(t) > 0 \Rightarrow a_{11} > 0, \text{ and } 4a_{12}^2 - 4a_{11}a_{22} = -4\det(A_f) < 0,$$
$$\varphi(t) > 0 \Leftrightarrow a_{11} > 0 \text{ and } \det(A_f) > 0.$$

which completes the proof. □

2.2 Optimization of Condense Discrete Functions

To obtain minimization results for a given condense discrete convex function, we require the given condense discrete convex function to be C^1. After defining the local and global minimum point concepts of condense discrete convex functions, we prove optimization results for C^1 condense discrete convex functions. Condense discrete concave function maximization results follow similarly.

We let $\bigcup_{i=1}^{\infty} S_i = \mathbb{Z}^n$ where S_i is a nonempty sufficiently small condense discrete convex neighborhood to support quadratic approximation of f, $\bigcap_{i=1}^{\infty} S_i = \emptyset$ and $\bigcap_{i \in I} S_i \neq \emptyset$ for all S_i where S_i have at least one common element for all $i \in I$, I is a finite index set, and $\{s_i\}$ is a singleton in $\mathbb{Z}^n$.

The partial derivative operator of a C^1 discrete funtion $f : \mathbb{Z}^n \to \mathbb{R}$ will be denoted by $\partial f(x) := \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n}\right)$.

Definition 2. The local minimum of a condense discrete C^1 function $f : \mathbb{Z}^n \to \mathbb{R}$ is the minimal value of f in a local neighborhood $\bigcup_{i \in I} S_i$ which is also the smallest value in a neighborhood $N = \bigcup_{j \in J} \left(\bigcup_{i \in I_j} S_i\right)$ where J is a finite index set. The global minimum value of a condense discrete convex function $f : \mathbb{Z}^n \to \mathbb{R}$ is the minimum value of f in the entire integer space $\mathbb{Z}^n$.

We define the set of local minimums of a C^1 condense discrete convex function f by

$$\Psi = \{\rho = (\rho_1, ..., \rho_n) : \rho_i \in \{\lceil \gamma_i \rceil, \lfloor \gamma_i \rfloor\} \subset \mathbb{Z} \text{ for all } i\} \subset \mathbb{Z}^n.$$

where $\partial f(\gamma) = 0$ holds for $\gamma \in \mathbb{R}^n$. As the domain is $\mathbb{Z}^n$, we consider the solutions in Ψ where $\rho_i = \lceil \gamma_i \rceil$ or $\rho_i = \lfloor \gamma_i \rfloor$ is the solution for multivariable integer function f.

Lemma 1. Let $f : \mathbb{Z}^n \to \mathbb{R}$ be a C^1 condense discrete convex function in $N \subset \mathbb{Z}^n$. Then there exists a local minimum value in $N \subset \mathbb{Z}^n$ such that

$$f_0 = \min_{\beta \in \Psi} \{f(\beta)\}.$$

Proof. Let $f : N \to \mathbb{R}$ be a C^1 strict condense discrete convex function. Therefore f has a local minimum value $f(x_0)$ in some neighborhood $S = \bigcup_{i \in I} S_i$ by

theorem 1. By definition of N, $\underset{i\in I}{\cup} S_i \subseteq N$ hence $f(x_0)$ is also the local minimum in the neighborhood N.

It is well known that the local minimum of a C^1 function f is obtained when the system of equations

$$\frac{\partial f(x)}{\partial x_i} = \lim_{t\to 0} \frac{f(x+te_i) - f(x)}{t} = 0$$

is solved simultaneously for all $i, 1 \le i \le n$. We first find $\partial f(x) = 0$ which implies the existence of a $\gamma_i \in \mathbb{R}$ for all i. Noting that the domain is $\mathbb{Z}^n$, we take the ceiling and floor of the components of γ_i to obtain the minimal point which consist of integer numbers $\lfloor \gamma_i \rfloor$ or $\lceil \gamma_i \rceil$ for all i. This gives a local minimum point $\beta \in \Psi$ and the corresponding value $f_0 = \min_{\beta\in\Psi} \{f(\beta)\}$. □

The following result for condense discrete convex functions is a result similar to a result in real convex analysis.

Theorem 2. Let $f : \mathbb{Z}^n \to \mathbb{R}$ be a C^1 strict condense discrete convex function. Then the set of local minimums of f form a set of global minimums and vice versa.

Proof. Suppose $f : \mathbb{Z}^n \to \mathbb{R}$ be a C^1 condense discrete convex function. Let $\overset{\infty}{\underset{i=1}{\cup}} S_i = \mathbb{Z}^n$ where S_i are sufficiently small condense discrete neighborhoods that support quadratic approximation of f for all i, and $\overset{\infty}{\underset{i=1}{\cap}} S_i = \emptyset$. Let Ω_1 be the set of local minimum points of f in $\mathbb{Z}^n$, and Ω_2 be the set of global minimum points of f in $\mathbb{Z}^n$.

Let $f : \mathbb{Z}^n \to \mathbb{R}$ be a C^1 condense discrete convex function and suppose f has global minimum points in $\mathbb{Z}^n = \overset{\infty}{\underset{i=1}{\cup}} S_i$. Noting that f is nonlinear, there exists a finite collection of S_i, $\underset{i\in I_0}{\cup} S_i$, where the global minimum points are located. The solution set of $\frac{\partial f(x)}{\partial x_j} = 0$ for all $j, 1 \le j \le n$, gives the set of local minimums in S_i. Therefore for all $x \in \Omega_2$ there exists a set of integer vectors $y \in \Omega_1$ such that $\min_{x\in\Omega_1} f(x) = f(y)$ which indicates $\Omega_2 \subset \Omega_1$ since $\underset{i\in I_0}{\cup} S_i \subset N \subset \mathbb{Z}^n$.

Now suppose there exists a vector x_0 in a local neighborhood $S = \underset{i\in I_1}{\cup} S_i$ such that $x_0 \notin \Omega_2$ (Note that x_0 is not necessarily an element of Ω_1 since it is a local minimum in a local setting). x_0 is a local minimum which is not a global minimum in S, therefore there exist x_1 and y_1 such that $f(x_0) > f(x_1) > f(y_1)$ in $N = \underset{j\in J}{\cup} \left(\underset{i\in I_j}{\cup} S_i \right) \supset S$ where y_1 becomes the new local minimum of the local neighborhood N. Therefore y_0 is the new local minimum of N where x_0 is not a local minimum of N. Suppose y_0 is a local minimum that is not a global minimum otherwise it would be an element of Ω_2. Continuing to enlarge the local obtained neighborhoods in this way to the entire space $\mathbb{Z}^n$, we obtain a set of points in a local neighborhood D of $\mathbb{Z}^n$ where local minimum points $x \in \Omega_1$ satisfy $f(x) < f(y)$ for all $y \in \mathbb{Z}^n - D$. Therefore $x \in \Omega_2$ and hence $\Omega_1 \subset \Omega_2$ which completes the proof. □

Next we consider an adaptation of Rosenbrook's function suggested by Yüceer [17] and show that this function is a condense discrete convex function.

3 An Example

Yüceer [17] shows that the adaptation of Rosenbrook's function

$$f(k,\mu) = 25(2\mu - k)^2 + \frac{1}{4}(2-k)^2 \quad \text{where } k,\mu \in \mathbb{Z}. \tag{1}$$

is not a discretely convex function when continuous variables are restricted to the integer lattice. Here, we first prove the condense discrete convexity of the function given in (1) and then show that the set of local minimums is also the set of global minimums.

The diagonal elements of the discrete Hessian matrix that corresponds to $f(k,\mu)$ are

$$\begin{aligned}
\nabla_{11} f(k,\mu) &= 25\,(2\mu - k - 2)^2 + \frac{1}{4}k^2 - 50\,(2\mu - k - 1)^2 \\
&\quad - \frac{1}{2}(1-k)^2 + 25\,(2\mu - k)^2 + \frac{1}{4}(2-k)^2 \\
&= \frac{101}{2} > 0. \\
\nabla_{22} f &= 25\,(2j + 4 - i)^2 - 50\,(2j + 2 - i)^2 + 25(2j - i)^2 \\
&= 200.
\end{aligned}$$

By the symmetry of the discrete Hessian matrix, the off diagonal elements of the discrete Hessian matrix are

$$\begin{aligned}
\nabla_{12} f &= \nabla_{21} f = 25(2j + 2 - i - 1)^2 - 25(2j - i - 1)^2 \\
&\quad - 25(2j + 2 - i)^2 + 25(2j - i)^2 \\
&= -100.
\end{aligned}$$

Therefore

$$\begin{aligned}
\det(H) &= 200.\frac{101}{2} - (100)^2 \\
&= 100.101 - (100)^2 \\
&= 100.
\end{aligned}$$

This indicates that the discrete Hessian matrix is positive definite. Therefore, the adaptation of the Rosenbrook's function given in the equality (1) (see Fig. 1.) is a strict condense discrete convex function.

Now we show that the set of local minimums of the adaptation of the condense discrete convex Rosenbrook's function is also the set of global minimums. Clearly, f is a C^1 function therefore

$$\partial f(k,\mu) = 0 \Rightarrow \begin{cases} \frac{\partial f}{\partial k} = -50(2\mu - k) - \frac{1}{2}(2-k) = 0 \\ \frac{\partial f}{\partial \mu} = 100(2\mu - k) = 0 \end{cases}$$

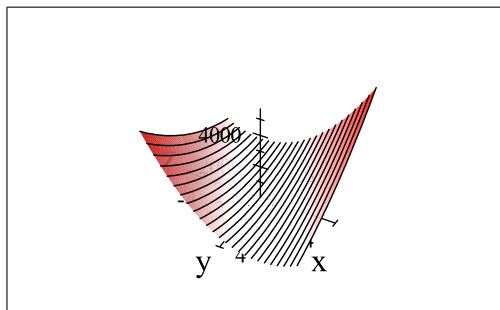


Fig. 1. An adaptation of Rosenbrook function suggested by Yüceer [13]

where simultaneous solution of this system of two equations indicate $k = 2$ and $\mu = 1$. Therefore the minimal value is $f(2, 1) = 0$. Since Rosenbrook's function is a C^1 condense discrete convex function, the local minimum point set which is the singleton $\{(2, 1, 0)\}$ is also the set of global minimum points.

An example of a function that is real convex but not condense discrete convex function is $g(x, y) = (x - y)^2$ since $\det(g) = 0$.

Acknowledgments. We would like to thank the reviewers for their constructive and helpful comments.

References

1. Denardo, E.V.: Dynamic Programming. Prentice-Hall, Englewood Cliffs (1982)
2. Favati, P., Tardella, F.: Convexity in Nonlinear Integer Programming. Ricerca Operativa 53, 3–44 (1990)
3. Fox, B.: Discrete optimization via marginal analysis. Management Sci. 13, 210–216 (1966)
4. Fujishige, S., Murota, K.: Notes on L-/M-convex functions and the separation theorems. Math. Prog. 88, 129–146 (2000)
5. Hirai, H., Murota, K.: M-convex functions and tree metrics. Japan J. Industrial Applied Math. 21, 391–403 (2004)
6. Kiselman, C.O., Christer, O.: Local minima, marginal functions, and separating hyperplanes in discrete optimization. In: Bhatia, R. (ed.) Abstracts: Short Communications; Posters. International Congress of Mathematicians, Hyderabad, August 19-27, pp. 572–573 (2010)
7. Kiselman, C.O., Acad, C. R.: Local minima, marginal functions, and separating hyperplanes in discrete optimization. Sci. Paris, Ser. I, (or Three problems in digital convexity: local minima, marginal functions, and separating hyperplanes - The case of two variables, by C.O. Kiselman, Manuscript) (2008)
8. Kiselman, C. O., Samieinia S.: Convexity of marginal functions in the discrete case. manuscript (2010), http://www2.math.uu.se/~kiselman/papersix.pdf
9. Miller, B.L.: On minimizing nonseparable functions defined on the integers with an inventory application. SIAM J. Appl. Math. 21, 166–185 (1971)

10. Moriguchi, S., Murota, K.: Discrete Hessian matrix for L-convex functions. IECE Trans. Fundamentals, E88-A (2005)
11. Murota, K.: Convexity and Steinitz's exchange property. Adv. Math., 272–311 (1996)
12. Murota, K.: Discrete convex analysis. Math. Prog. 83, 313–371 (1998)
13. Murota, K.: Discrete convex analysis. Society for Industrial and Applied Mathematics, Philadelphia (2003)
14. Murota, K., Shioura, A.: M-convex function on generalized polymatroid. Math. Oper. Res. 24, 95–105 (1999)
15. Rockafellar, R.T.: Convex Analysis. Princten University Press, Princeton (1970)
16. Ui, T.: A note on discrete convexity and local optimality. Japan J. Indust. Appl. Math. 23, 21–29 (2006)
17. Yüceer, U.: Discrete convexity: convexity for functions defined on discrete spaces. Disc. Appl. Math. 119, 297–304 (2002)

Path Trading: Fast Algorithms, Smoothed Analysis, and Hardness Results

André Berger[1], Heiko Röglin[2], and Ruben van der Zwaan[1]

[1] Maastricht University, The Netherlands
{a.berger,r.vanderzwaan}@maastrichtuniversity.nl
[2] University of Bonn, Germany
heiko@roeglin.org

Abstract. The Border Gateway Protocol (BGP) serves as the main routing protocol of the Internet and ensures network reachability among autonomous systems (ASes). When traffic is forwarded between the many ASes on the Internet according to that protocol, each AS selfishly routes the traffic inside its own network according to some internal protocol that supports the local objectives of the AS. We consider possibilities of achieving higher global performance in such systems while maintaining the objectives and costs of the individual ASes. In particular, we consider how *path trading*, i.e. deviations from routing the traffic using individually optimal protocols, can lead to a better global performance. Shavitt and Singer ("Limitations and Possibilities of Path Trading between Autonomous Systems", INFOCOM 2010) were the first to consider the computational complexity of finding such path trading solutions. They show that the problem is weakly NP-hard and provide a dynamic program to find path trades between pairs of ASes.

In this paper we improve upon their results, both theoretically and practically. First, we show that finding path trades between sets of ASes is also strongly NP-hard. Moreover, we provide an algorithm that finds all Pareto-optimal path trades for a pair of two ASes. While in principal the number of Pareto-optimal path trades can be exponential, in our experiments this number was typically small. We use the framework of smoothed analysis to give theoretical evidence that this is a general phenomenon, and not only limited to the instances on which we performed experiments. The computational results show that our algorithm yields far superior running times and can solve considerably larger instances than the previous dynamic program.

1 Introduction

The Border Gateway Protocol (BGP) serves as the main routing protocol on the top level of the Internet and ensures network reachability among autonomous systems (ASes). When traffic is forwarded from a source to a destination, these ASes cooperate in order to provide the necessary infrastructure needed to ensure the desired services. However, ASes do also compete and therefore follow their individual strategies and policies when it comes to routing the traffic within

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 43–53, 2011.
© Springer-Verlag Berlin Heidelberg 2011

their own network. Such locally preferable routing decisions can be globally disadvantageous. Particularly, the way how one AS forwards traffic and through which ingress node another AS may therefore receive the traffic can make a huge difference in the costs for that other AS. Behaving selfishly usually means that an AS routes its traffic according to the least expensive route, also known as hot-potato routing, without regarding the costs of the next AS in the BGP path. This is supported by strong evidence by Teixeira et al. [17].

Quite a number of protocols have been suggested that require the exchange of information and coordination in order to overcome global suboptimality while at the same time improving the costs for each individual AS [6,7,18]. Recently, Shavitt and Singer [13] considered the case where ASes might be willing to *trade* traffic in such a way that the costs for both ASes do not increase w.r.t. the hot-potato routing, and term this problem *path trading*. They prove that for two ASes the problem of deciding whether there is a feasible path trade is weakly NP-hard. Further, they develop an algorithm based on dynamic programming to find the "best" trading between a pair. Lastly, they give experimental evidence that path trading can have benefits to autonomous systems.

In this paper we extend their work in the following way. We show that path trading is also strongly NP-hard when an arbitrary number of ASes is considered. This justifies the approach taken by Shavitt and Singer as well as the approach taken in this paper to concentrate on path trades between pairs of ASes. We propose a new algorithm for finding path trades between pairs of ASes that is based on the concept of *Pareto efficiency*. We have implemented both, our algorithm and the algorithm of Shavitt and Singer, and tested them on real Internet instances stemming from [12]. Besides the added advantage that our algorithm obtains *all* Pareto-optimal path trades, it is very fast and has low memory consumption. As the problem is NP-hard, we cannot expect that the algorithm performs well on all possible inputs. However, in order to support the experimental results we consider our algorithm in the framework of *smoothed analysis*, which was introduced in 2001 by Spielman and Teng [15] to explain why many heuristics with a bad worst-case performance work well on real-world data sets. We show that even though there are (artificial) worst-case instances on which the heuristic performs poorly, it has a polynomial expected running time on instances that are subject to small random perturbations.After its introduction, smoothed analysis has been applied in many different contexts (see [16] for a nice survey).

Finding path trades can be viewed as an optimization problem with multiple objectives that correspond to the costs of the different ASes. A *feasible path trade* is then a solution that is in every objective at least as good as the hot-potato routing. We say that such a path trade *dominates* the hot-potato routing if it is strictly better in at least one objective. This brings us to the well-known concept of *Pareto efficiency* or *Pareto optimality* in multiobjective optimization: A solution is called *Pareto-optimal* if it is not dominated by any other solution, that is, a solution is Pareto-optimal if there does not exist another solution that is at least as good in all criteria and strictly better in at least one criterion. We call the set of Pareto-optimal solutions *Pareto set* or *Pareto curve* for short.

Then the question of whether there is a feasible path trade can simply be formulated as the question whether the hot-potato routing is Pareto-optimal or not. This immediately suggests the following algorithm to find a feasible path trade: Enumerate the set of Pareto-optimal solutions, and then either output that there is no path trade if the hot-potato routing belongs to the Pareto set, or output a Pareto-optimal solution that dominates the hot-potato routing if it is not Pareto-optimal. Also, finding the Pareto set gives the flexibility to choose a solution based on preference. While some solutions might offer great global gain, these trade-offs might be unreasonable from a fairness perspective.

The aforementioned algorithm only works when the Pareto set is small because otherwise the computation becomes too time consuming. Our experiments show that the number of Pareto-optimal path trades is indeed small and that despite the NP-hardness result by Shavitt and Singer we can solve this problem efficiently in practice for two ASes.

For path trading between an arbitrary number of ASes, however, there is little hope for obtaining such a result: We show that our strong NP-hardness result implies that this problem cannot be solved efficiently even in the framework of smoothed analysis.

Related Work. The potential benefits of collaboration between neighboring ASes and the necessary engineering framework were first introduced by Winick et al. [18]. They consider the amount of information that needs to be shared between the ASes in order to perform mutually desirable path trades and how to limit the effect of path trades between neighboring ASes on the global flow of traffic. The first heuristics for path trading to improve the hot-potato routing were evaluated by Majahan et al. [7]. Majahan et al. also developed a routing protocol that provides evidence that path trading can improve global efficiency in Internet routing. Other related work in the area of improving the global performance while maintaining the objectives of the different ASes has been done by Yang et al. [19], Liu and Reddy [6], and by Quoitin and Bonaventure [9]. Since ASes usually compete, one cannot expect them to reveal their complete network and cost structure when it comes to coordinating the traffic between the ASes. This aspect is considered in the work by Shrimali et al. [14], using aspects from cooperative game theory and the idea of Nash bargaining. Goldenberg et al. [4] develop routing algorithms in a similar context to optimize global cost and performance in a multihomed user setting, which extends previous work in that area [1,3,11].

2 Model and Notation

The model is as follows. We have the Internet graph $G = (V, E)$, where every vertex represents a point/IP-address. Further, there are k ASes and the vertex set V is partitioned into mutually disjoint sets $V_1, \ldots, V_k$, where V_i represents all points in AS i. We denote by E_i all edges within AS i, that is, the set of edges E is partitioned into $E_1, \ldots, E_k$, and the set of edges between different ASes. The graph G is undirected, and each edge $e \in E$ has a length $\ell(e) \in \mathbb{R}_{\geq 0}$.

The traffic is modeled by a set of *requests* R, where each request is a triple (s, t, c), where $s \in V$ and $t \in V$ are source and sink nodes, respectively, and $c \in \mathbb{R}_{\geq 0}$ is the cost of the corresponding request. The BGP protocol associates with each request a sequence of ASes which specifies the order in which the request has to be routed through the ASes. Since most of the paper is about the situation between *two* ASes, we leave this order implicit. The cost of routing a request with cost c through edge e is $\ell(e) \cdot c$. For simplicity, the costs of routing a packet between two ASes are assumed to be zero, but each request can be routed at most once from an AS to the next AS. The input for PATH TRADING consists of the graph G and requests as described previously. We denote by n the number of nodes in V.

For a given graph G and a request (s, t, c) we say that a path P is *valid* if it connects s to t and visits the ASes in the order that is associated with this request by the BGP protocol. This means, in particular, that every valid path goes through every AS at most once. A solution to PATH TRADING is a mapping that maps each request $(s, t, c) \in R$ to a valid path from s to t in graph G. Let us assume that the requests in R are $(s_1, t_1, c_1), \ldots, (s_r, t_r, c_r)$ and that the paths $P_1, \ldots, P_r$ have been chosen for these requests. Then AS i incurs costs on all edges in E_i, i.e., it incurs a total cost of

$$\sum_{j=1}^{r} \Big(c_j \cdot \sum_{e \in P_j \cap E_i} \ell(e) \Big). \tag{1}$$

The *hot-potato route* of a request (s, t, c) is defined to be the concatenation of shortest path routes for the ASes it goes through. To be precise, assume that the BGP protocol associates the route $i_1, \ldots, i_m$ with $s \in V_{i_1}$ and $t \in V_{i_m}$ with this request. Then AS i_1 sends the request from s to the vertex $s_2 \in V_{i_2}$ that is closest to s along the shortest path. Then AS i_2 sends the request from s_2 to the vertex $s_3 \in V_{i_3}$ that is closest to s_2 along the shortest path, and so on. The complete hot-potato route for request (s, t, c) is then the concatenation of these paths. Note that the hot-potato route is not necessarily unique, and in the following we will assume that some hot-potato route is chosen for each request.

Consequently, the costs of the hot-potato routing that an AS i incurs are equal to Equation 1, where the paths $P_1, \ldots, P_r$ are the hot-potato paths. We call a solution to PATH TRADING a *path trade* and if the costs for all involved ASes are less or equal to their hot-potato costs, then we call it a *feasible path trade*. In the following, let $[n]$ be the set of integers $\{1, ..., n\}$. For a vector $x \in \mathbb{R}^n$, let x_i be the i-th component of x.

Due to space limitations the proofs of the complexity results (Theorems 1, 2, 4 and Corollary 2) are deferred to full version of this paper.

3 Complexity Results and Smoothed Analysis

Our first result is about the complexity of PATH TRADING and extends the weak NP-hardness result of Shavitt and Singer [13]. The proof uses a reduction from 3-PARTITION.

Theorem 1. *Finding a feasible path trade apart from the hot-potato routing is strongly NP-hard.*

Given the above theorem, in order to develop fast algorithms, we concentrate on path trading between two ASes, and will now present our algorithm for this case. As mentioned before, this algorithm is based on the concept of Pareto efficiency and it enumerates all Pareto-optimal path trades. In the worst case the number of Pareto-optimal solutions can be exponential, but our experiments suggest that on real-world data usually only a few solutions are Pareto-optimal. To give a theoretical explanation for this, we apply the framework of smoothed analysis. The algorithm is a dynamic program that adds the requests one after another, keeping track of the Pareto-optimal path trades of those requests that have already been added.

For a request (s, t, c), a path P from s to t, and $i \in \{1, 2\}$, we denote by $C_i(P)$ the costs incurred by AS i due to routing the request along path P. To keep the notation simple, assume in the following discussion w.l.o.g. that $s \in V_1$ and $t \in V_2$. We denote by $\mathcal{P}(s, t)$ the set of all Pareto-optimal valid paths from s to t. Remember that a path is valid if it starts at s, terminates at t, and does not go back to V_1 after leaving V_1 for the first time. Such a path P belongs to $\mathcal{P}(s, t)$ if there does not exist another valid path that induces strictly lower costs for one AS and not larger costs for the other AS than P. We assume that in the case that there are multiple paths that induce for both ASes exactly the same costs, only one of them is contained in $\mathcal{P}(s, t)$.

Let $P \in \mathcal{P}(s, t)$ be some Pareto-optimal path and let $v \in V_1$ be the *boundary node* at which the path leaves AS 1. Then the subpaths from s to v and from v to t must be shortest paths in AS 1 and AS 2, respectively. Otherwise, P cannot be Pareto-optimal. Hence, the number of Pareto-optimal paths in $\mathcal{P}(s, t)$ is bounded from above by the number of boundary nodes of AS 1 that connect to AS 2. For each pair $s \in V_1$ and $t \in V_2$, the set $\mathcal{P}(s, t)$ can be computed in polynomial time.

Our algorithm first computes the set $\mathcal{P}_1$ of Pareto-optimal path trades for only the first request (s_1, t_1, c_1). This is simply the set $\mathcal{P}(s_1, t_1)$. Based on this, it computes the set $\mathcal{P}_2$ of Pareto-optimal path trades for only the first two requests, and so on. Thus the elements in $\mathcal{P}_i$ are tuples $(P_1, \ldots, P_i)$ where each P_j is a valid path for the jth request.

Algorithm 1. Algorithm to compute the Pareto set

$\mathcal{P}_1 = \mathcal{P}(s_1, t_1)$;
for $i = 2$ to r **do**
  $\mathcal{P}_i = \{(P_1, \ldots, P_i) \mid (P_1, \ldots, P_{i-1}) \in \mathcal{P}_{i-1}, P_i \in \mathcal{P}(s_i, t_i)\}$;
  Remove all solutions from $\mathcal{P}_i$ that are dominated by other solutions from $\mathcal{P}_i$.
  If $\mathcal{P}_i$ contains multiple solutions that induce for both ASes exactly the same costs, then remove all but one of them.
end for
Return $\mathcal{P}_r$

Theorem 2. *For $i \in [r]$, the set $\mathcal{P}_i$ computed by Algorithm 1 is the set of Pareto-optimal path trades for the first i requests. In particular, the set $\mathcal{P}_r$ is the set of Pareto-optimal path trades for all requests. Algorithm 1 can be implemented to run in time $O(n \log n \cdot \sum_{i=1}^{r} |\mathcal{P}_i| + nr|E| \log n)$.*

We start by reviewing a result due to Beier et al. [2] who analyzed the number of Pareto-optimal solutions in binary optimization problems with two objective functions. They consider problems whose instances have the following form: the set of feasible solutions S is a subset of $\{0, \dots, F\}^n$ for some integers F and n, and there are two objective functions $w^{(1)} : S \to \mathbb{R}$ and $w^{(2)} : S \to \mathbb{R}$ that associate with each solution $x \in S$ two weights $w^{(1)}(x)$ and $w^{(2)}(x)$ that are both to be minimized. While $w^{(2)}$ can be an arbitrary function, it is assumed that $w^{(1)}$ is linear of the form $w^{(1)}(x) = w_1 x_1 + ... + w_n x_n$.

In a worst-case analysis, the adversary would be allowed to choose the set of feasible solutions S, and the two objective functions $w^{(1)}$ and $w^{(2)}$. Then it can easily be seen that there are choices such that the number of Pareto-optimal solutions is exponential. To make the adversary less powerful and to rule out pathological instances, we assume that the adversary cannot choose the coefficients $w_1, \dots, w_n$ exactly. Instead he can only specify a probability distribution for each of them according to which it is chosen independently of the other coefficients. Without any restriction, this would include deterministic instances as a special case, but we allow only probability distributions that can be described by a density function that is upper bounded by some parameter $\phi \geq 1$.

We denote by $f_i : \mathbb{R}_{\geq 0} \to [0, \phi]$ the probability density according to which w_i is chosen, and we assume that the expected value of w_i is in $[0, 1]$.

Theorem 3 (Beier et al., [2]). *For any choice of feasible solutions $\mathcal{S} \subseteq \{0, ..., F\}^n$, any choice of $w^{(2)}$ and any choice of density functions $f_1, \dots, f_n$, the expected number of Pareto-optimal solutions is bounded by $O(\phi n^2 F^2 \log F)$.*

Now we formulate our problem in terms of Theorem 3. For this, we assume that all requests have positive integer costs. Let F denote an upper bound on the maximal costs possible on any edge, e.g., $F = \sum_{(s,t,c) \in R} c$. Let $m = |E|$ and assume that the edges in E are ordered arbitrarily. Then each path trade leads to a cost vector $x \in \{0, \dots, F\}^m$ where x_1 denotes the total cost of all requests that use the first edge in E, x_2 denotes the total cost of all requests that use the second edge in E, and so on. If two solutions lead to the same cost vector, then it suffices to remember one of them, and hence, we can assume that the set of possible path trades can essentially be described by the set $S \subseteq \{0, \dots, F\}^m$ of possible cost vectors. Given such a cost vector $x \in \{0, \dots, F\}^m$, we can express the cost $w^{(1)}(x)$ of the first AS simply as $\sum_{e \in E_1} \ell(e) x_e$. The costs of the second AS can be defined analogously, and so it looks that Theorem 3 directly applies when we perturb all edge lengths $\ell(e)$ of edges $e \in E_1$ as these edge lengths are the coefficients in the linear objective function $w^{(1)}$. However, there is a small twist. In Theorem 3 all coefficients in the linear objective function $w^{(1)}$ are chosen randomly. Our objective function, however, does not contain terms for the edges $e \in E_2$. Or with other words the coefficients are 0 for these edges. If we apply

Theorem 3 directly, then also these zero coefficients would be perturbed, which would destroy the combinatorial structure of the problem as then suddenly the cost of the first AS would depend on what happens on edges $e \in E_2$.

To avoid this side effect, we remodel the feasible region S. As argued before, each solution leads to a cost vector $x \in \{0,\ldots,F\}^m$, but now we care only about the part of the vector for E_1. Let us define $m' = |E_1| \le m$. Then each solution leads to a cost vector $x \in \{0,\ldots,F\}^{m'}$ that contains only the costs of the first AS. Now of course different solutions can lead to the same vector x if they differ only in the way how the traffic is routed in the second AS. However, Theorem 3 allows completely general objective functions $w^{(2)}$, which we exploit by defining $w^{(2)}(x)$ for a vector $x \in \{0,\ldots,F\}^{m'}$ to be the smallest cost for the second AS that can be achieved with any solution whose cost vector for the first AS results in x. This formulation implies the following corollary.

Corollary 1. *Given a path trading instance in which the edge lengths $\ell(e)$ for all $e \in E_1$ are randomly chosen according to probability distributions that satisfy the same restrictions as those in Theorem 3, the expected number of Pareto-optimal solutions is bounded by $O(\phi m^2 F^2 \log F)$.*

Given that the expected number of Pareto-optimal solutions is small, we still have to show that Algorithm 1 computes the Pareto curve in expected polynomial time. This will be established by the following Corollary.

Corollary 2. *Algorithm 1 computes the Pareto curve in expected time $O(\phi n m^2 \log n \cdot r F^2 \log F)$.*

The reason that we concentrate our efforts on path trading between two ASes was the hardness result in Theorem 1. We can extend this result and also show that there is no hope for Path Trading with an arbitrary number of ASes to be solvable efficiently in the framework of smoothed analysis.

Theorem 4. *There is no smoothed polynomial time algorithm for* Path Trading *with an arbitrary number of ASes, unless $NP \subseteq BPP$.*

4 Evaluation

In this section we present the experimental results about the performance of our algorithm on the IP-level Internet graph from DIMES [12]. We compare it, in particular, with the performance of the dynamic program used by Shavitt and Singer, and we answer the following questions:

- *How fast can we compute the Pareto curve?* Algorithm 1 is very fast and scales well.
- *How robust are both algorithms?* When we add random costs to all requests, the running time of Algorithm 1 does not increase much. This suggests even in environments with large costs, Algorithm 1 will perform well. The running time of the dynamic program directly depends on the costs, and it becomes quickly infeasible to compute solutions for even small instances.

– *How many ASes are involved in path trading?* In our experiments we see that for low amount of requests, roughly 60% of all ASes engage in path trading.

The answers to these questions are, of course, depending on the assumptions we made. As Shavitt and Singer, we assume that traffic is symmetric, i.e., the number of requests sent from AS A to AS B is the same as the number of requests sent from AS B to AS A for every pair of ASes. This assumption is not necessarily true, but it is common and used, e.g., in [13], [7], and [14]. One could imagine that in real networks requests are not evenly spread, but are more concentrated between popular ASes for example. Still, even for low amounts of requests there was substantial gain for the ASes involved. This indicates that even if the traffic between two ASes is asymmetric, they have a good chance of gaining from path trading as long as there is non-zero traffic in both directions. Our second assumption is that each request between two ASes has to be routed between two nodes that are chosen uniformly at random from these ASes. Our third assumption is that all edges have length 1, i.e., the number of hops is used to measure the costs of an AS for routing a request. By absence of real data, we feel that this a reasonable and common assumption. We first perform experiments in which every request has costs 1 and then repeat the experiments with requests with randomly chosen costs. These experiments demonstrate that our method is robust against changes of the costs of the requests. In the following subsection we present the details of the experimental setup. The algorithm by Shavitt and Singer is named Algorithm 2. Then we show and discuss the experimental results.

4.1 Experimental Setup

We assume that traffic is symmetric. We used the Internet graph from DIMES [12], and we assumed that every edge length is one, and all packets have cost one. So, the costs of a request are the number of hops on the route. The whole Internet graph from DIMES contains roughly 27 thousand ASes and 3.5 million nodes. Of all ASes, 1276 ASes and 4348 AS pairs were sufficiently connected: These pairs had edges between them and more than one boundary node. To determine participation, we simulated a low number of requests for each sufficiently connected pair, to find out whether a small or a large fraction of ASes are involved in path trading.

For both algorithms, we need to calculate shortest paths beforehand. Because of the large number of possible routings, many shortest paths need to be computed. This was all done as part of the preprocessing, and all shortest paths were stored in a hash-table for fast access for both algorithms. In the following, this time is not included in the running times of the algorithms.

To measure how many ASes could benefit from path trading, we simulated 5 requests for each of the 4348 sufficiently connected AS pairs in either direction. For comparing performance and robustness we selected a subset of 15 AS pairs arbitrarily among the AS pairs that benefited from path trading in the first experiment where 5 requests were sent in either direction.

To get some idea about how robust both algorithms are, we increased the costs of the requests. For each request (s, t, c) we set c as $c = 1 + X$, where X is a random variable, normally distributed with mean 0 and standard deviation σ. Further, X was capped at 0 from below and at 10 from above for $\sigma \in \{1, 2, 3, 4, 5\}$. For $\sigma = 10$, X was capped at 0 from below and at 20 from above. This was done to prevent extremely long running times for the dynamic program. For $\sigma \in \{15, 20, 25, 50\}$, X was only capped at 0 from below. All numbers were rounded to the nearest integer. We simulated 10 requests in either direction, for each of the pairs.

4.2 Experimental Results

Performance. Table 1 shows a comparison of the running times of Algorithm 1 and Algorithm 2. The running times are the total of the running time over the 15 selected pairs in seconds. In these ASes, roughly 37% of the costs were saved by path trading. As can be seen, the running time of Algorithm 2 quickly becomes very high.

The memory usage is dominated by the number of Pareto optimal solutions, and each Pareto optimal solution is represented as a tuple of two integers. Figure 1(a) shows a graphical comparison of both algorithms. Not only is Algorithm 1 fast for small amounts of requests, it can handle up to ten times more requests in the same time as Algorithm 2.

Robustness. We find that the running time of both Algorithm 1 and Algorithm 2 is influenced by larger request costs, but not to the same degree. Figure 1(a) shows the running times of both algorithms. As can be seen, the running time of Algorithm 2 quickly spirals out of control. Algorithm 1 stays computable, although the running time does increase. Figure 1(b) displays the running times normalized with regard to the running time without perturbations for both Algorithm 1 and 2.

The steep increase of the running time of Algorithm 2 comes at no surprise as the dynamic program is directly dependent on the costs of the routing, and not on the number of different choices or the complexity of the network. The experiments show that our algorithm is significantly more robust against non-uniform request costs.

Table 1. The performance of Algorithm 1 compared to Algorithm 2

# Requests	Algorithm 1 (s)	Algorithm 2 (s)	Ratio
1	0.02	0.09	1: 4.5
5	0.19	6.04	1: 31.79
10	1.09	84.31	1: 77.35
15	2.38	270.87	1:113.81
19	4.01	519.27	1:129.49

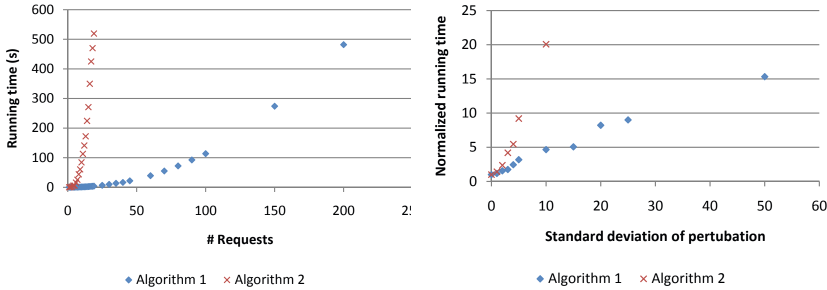


Fig. 1. (a) Running times of both algorithms compared. (b) The normalized running times of both algorithms plotted against magnitude of the perturbations.

Acknowledgements. The authors would like to thank Tobias Brunsch for proof-reading this manuscript and for his helpful comments.

References

1. Akella, A., Maggs, B., Seshan, S., Shaikh, A., Sitaraman, R.: A measurement-based analysis of multihoming. In: SIGCOMM, pp. 353–364 (2003)
2. Beier, R., Röglin, H., Vöcking, B.: The smoothed number of pareto optimal solutions in bicriteria integer optimization. In: Fischetti, M., Williamson, D.P. (eds.) IPCO 2007. LNCS, vol. 4513, pp. 53–67. Springer, Heidelberg (2007)
3. Dai, R., Stahl, D.O., Whinston, A.B.: The economics of smart routing and quality of service. In: Stiller, B., Carle, G., Karsten, M., Reichl, P. (eds.) NGC 2003 and ICQT 2003. LNCS, vol. 2816, pp. 318–331. Springer, Heidelberg (2003)
4. Goldenberg, D.K., Qiu, L., Xie, H., Yang, Y.R., Zhang, Y.: Optimizing cost and performance for multihoming. In: SIGCOMM, pp. 79–82 (2004)
5. Knuth, D.: The Art of Computer Programming, 3rd edn. Sorting and Searching, vol. 3. Addison-Wesley, Reading (1997)
6. Liu, Y., Reddy, A.L.N.: Multihoming route control among a group of multihomed stub networks. Computer Comm. 30(17), 3335–3345 (2007)
7. Mahajan, R., Wetherall, D., Anderson, T.: Negotiation-based routing between neighboring ISPs. In: NSDI, pp. 29–42 (2005)
8. Nemhauser, G.L., Ullmann, Z.: Discrete dynamic programming and capital allocation. Management Science 15(9), 494–505 (1969)
9. Quoitin, B., Bonaventure, O.: A cooperative approach to interdomain traffic engineering. In: EuroNGI (2005)
10. Röglin, H., Teng, S.-H.: Smoothed Analysis of Multiobjective Optimization. In: FOCS, pp. 681–690 (2009)
11. Sevcik, P., Bartlett, J.: Improving user experience with route control. Technical Report NetForecast Report 5062, NetForecast, Inc. (2002)
12. Shavitt, Y., Shir, E.: DIMES: let the Internet measure itself. ACM SIGCOMM Computer Communication Review 35(5), 71–74 (2005)
13. Shavitt, Y., Singer, Y.: Limitations and Possibilities of Path Trading between Autonomous Systems. In: INFOCOM (2010)
14. Shrimali, G., Akella, A., Mutapcic, A.: Cooperative interdomain traffic engineering using nash bargaining and decomposition. In: INFOCOM, pp. 330–338 (2007)

15. Spielman, D.A., Teng, S.-H.: Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. Journal of the ACM 51(3), 385–463 (2004)
16. Spielman, D.A., Teng, S.-H.: Smoothed analysis: an attempt to explain the behavior of algorithms in practice. Communic. of the ACM 52(10), 76–84 (2009)
17. Teixeira, R., Shaikh, A., Griffin, T., Rexford, J.: Dynamics of hot-potato routing in IP networks. In: SIGMETRICS, pp. 307–319 (2004)
18. Winick, J., Jamin, S., Rexford, J.: Traffic engineering between neighboring domains. Technical report (2002)
19. Yang, Y.R., Xie, H., Wang, H., Silberschatz, A., Krishnamurthy, A., Liu, Y., Li, E.L.: On route selection for interdomain traffic engineering. IEEE Network 19(6), 20–27 (2005)

Hierarchical Delaunay Triangulation for Meshing

Shu Ye and Karen Daniels

Department of Computer Science, University of Massachusetts, Lowell
{sye,kdaniels}@cs.uml.edu

Abstract. This paper discusses an elliptical pad structure and its polygonal approximation. The elliptical pad is a part of via model structures, which are important and critical components on today's multilayered Printed Circuit Board (PCB) and electrical packaging. To explore meshing characterization of the elliptical pad helps mesh generation over 3D structures for electromagnetic modeling (EM) and simulation on PCB and electrical packaging. Because elliptical structures are often key PCB features, we introduce a hierarchical mesh construct and show that it has several useful Delaunay quality characteristics. Then we show experimentally that Computational Geometry Algorithm Library's (CGAL) meshing of an elliptical structure at different resolution levels and with various aspect ratios produces patterns similar to our construct. In particular, our experiment also shows that the result of meshing is not only constrained Delaunay triangulation but also Delaunay triangulation.

Keywords: constrained Delaunay triangulation, mesh generation, CGAL.

1 Introduction

In recent years, the interconnect modeling on multilayered PCB and in packaging has become a bottleneck for successful high-speed circuit design [14]. The signal integrity issues, such as the signal propagation time, the digital pulse distortion, and the cross-talk, all affect the quality of the digital signal and can cause integrated circuit gate misswitching and introduce large bit rate error [12]. Therefore, simple physical constraints on the routing rules are no longer sufficient. For critical nets, accurate circuit simulation is needed, which requires accurate EM characterization on interconnects. The finite element based full-wave EM field solver can be applied to perform such tasks which, rely heavily on the quality of the finite element mesh generation [13]. Mesh generation for finite elements has been widely studied (see [4] for a survey). Techniques for mesh generation have been studied extensively in the geometric modeling and computational geometry communities [1,4,5]. Geometric and topological underpinnings of mesh generation are explored in [3]. In some cases (e.g. [9]), mesh generation is tightly coupled with the EM simulation method.

A via [6, 10] structure is heavily used in today's PCB and packaging. A via vertically connects different layers on the PCB. Fig. 1(a) depicts a coupled via structure. Each via in Fig. 1 consists of several features: 1) a cylindrical drill that extends through a hole in layer(s), 2) a small cylindrical pad at each end of the drill, and 3) a trace that extends from each pad to connect the via to the associated layer.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 54–64, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Fig. 1(b) shows triangle meshing for single via structure. In this paper we focus on the 2D elliptical pad part of via structures, which is approximated by a polygon for meshing.

Fig. 1. (a) Coupled “through hole” via structure (b) 2D meshing for single via structure (Courtesy of Cadence Design Systems)

Our high-level algorithm for generating meshing on structures of PCB and packaging: projects 3D structures orthogonally onto the *x-y* plane, generates 2D triangular meshing based on 2D projected datasets, and finally extrudes 2D triangles vertically through the PCB/packaging layers to form 3D prism meshing. The approach applies CGAL’s constrained Delaunay triangulation and inputs triangle element control criteria to control edge length and the angle of the mesh triangle element in the desired computational space [2]. This extrusion approach was introduced in [15]. We used it in [16], for only single (not coupled) serpentine line features.

The remainder of this paper is organized as follows. Section 2 briefly discusses triangle quality measures since they will be used to evaluate the method used in this paper. Section 3 discusses the ideas of Delaunay triangulation and constrained Delaunay triangulation. Section 4 focuses on describing our hierarchical Delaunay triangulation. Section 5 shows quality measures and mesh size trend for pad refinement, which is closely related to our proofs in Section 4. The experimental meshing results with various aspect ratios show that meshing produces not only a constrained Delaunay triangulation, but also a Delaunay triangulation. Finally, Section 6 concludes the paper and suggests directions for future work.

2 Quality Measure

The quality of triangulation directly affects the accuracy of the EM computation accuracy with FEM. This is true especially for full-wave EM modeling [11]. Various quality measures appear in the literature; see [4] for a survey. Since we focus on analyzing triangulation characterization on a 2D elliptical structure, we are only interested in measuring quality in 2D. In 2D we measure:

- number of triangles: this should be as small as possible to reduce FEM computation time;
- triangle angles: these should be as large as possible to avoid FEM simulation difficulties;

– a ratio involving triangle area and edge lengths: triangles should be as close to equilateral as possible.

The ratio we use is based on [7]. The element quality ratio q_t for a triangle is:

$$q_t = \frac{4\sqrt{3}A}{h_1^2+h_2^2+h_3^2}. \tag{1}$$

where A denotes the area, and h_1, h_2, and h_3 are the edge lengths. It is straightforward to show that q_t =1 for an equilateral triangle.

3 Delaunay Triangulation and Constrained Delaunay Triangulation

Delaunay triangulation [3, 5, 8, 13] provides a good foundation to support high-quality meshing in our context. We use the 2D Delaunay triangulation. This has the empty circle property: each triangle's circumcircle's interior contains no vertices. The Delaunay triangulation also maximizes the minimum triangle angle size, which supports our 2D quality criteria. Because we must include edges of structural features in the triangulation, we use a *constrained* Delaunay triangulation. Guaranteed inclusion of these edges typically implies sacrificing some mesh quality and adding extra vertices (Steiner points). Following [3], if we denote by E the set of edges of structural features that we must preserve, let *int* refer to interior, and let points $x, y \in R^2$ be visible from each other in a triangulation of E when $xy \notin E$ and $int\ xy \cap uv = \emptyset$, $\forall uv \in E$, then (assuming general position), an edge *ab*, with *a* and *b* both in the triangulation, belongs to the constrained Delaunay triangulation of *E* if:

(i) $ab \in E$, or;

(ii) *a* and *b* are visible from each other and there is a circle passing through *a* and *b* such that each triangulation vertex (not on *ab*) inside this circle is invisible from every point $x \in int\ ab$.

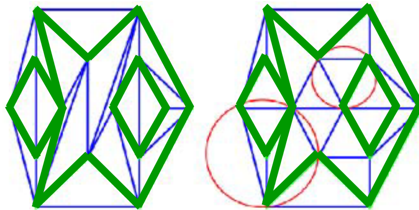


Fig. 2. (a): A constrained triangulation. (b): A constrained Delaunay triangulation. (courtesy of CGAL documentation [2], but with constrained edges of *E* thickened).

A constrained Delaunay triangulation satisfies this *constrained empty circle* property [2]. Said differently, "it is convenient to think of constrained edges as blocking the view. Then, a triangulation is constrained Delaunay if and only if the circumscribing circle of any facet encloses no vertex visible from the interior of the facet" [2]. It is shown in [3] that, among all constrained triangulations of *E*, the constrained Delaunay triangulation

maximizes the minimum angle. Fig. 2 above illustrates the constrained empty circle property of a constrained Delaunay triangulation, where thick segments are the constrained edges.

CGAL provides easy access to efficient and reliable geometric algorithms in a C++ library. It not only offers data structures and algorithms such as Delaunay triangulation and mesh generation which we require for our project, but also provides various geometric functions that we use elsewhere, such as Voronoi diagrams, Boolean operations on polygons and polyhedra, and shape analysis.

4 Hierarchical Delaunay Triangulation for Mesh Generation

Here we first show that when a circular pad (which is a special case of elliptical shape) is approximated by either an equilateral triangle or a regular hexagon, then existence of a Delaunay triangulation containing all the edges of the approximation to the pad is guaranteed if we allow insertion of Steiner points. In these simple base cases, there exist constrained Delaunay triangulations that are also Delaunay triangulations. Referring to Fig. 3, in Fig. 3(a) there is a single equilateral triangle. This triangle at level 1 of the refinement is trivially Delaunay and its quality according to Equation 1 is equal to one. In Fig. 3(b), the circle is approximated by a hexagon; this is level 2 of the hierarchy. If we let the length of a side of the hexagon equal 1, then we can triangulate the hexagon using 6 equilateral triangles that have a common vertex at the center of the circle that circumscribes the hexagon. Each of these 6 triangles has the empty circle property because each of their radii has length $1/\sqrt{3}$. For each such a triangle *t*, all remaining triangulation vertices are outside of *t*'s circumcircle because their distance from the center of *t*'s circumcircle exceeds $1/\sqrt{3}$ (because the radius of each adjoining triangle's circumcircle equals $1/\sqrt{3}$). Due to this empty circle property, the triangulation is not only constrained (because the hexagon edges are preserved), but also Delaunay. Because the triangles are all equilateral, we again have quality of 1 from Equation 1 (in Section 2).

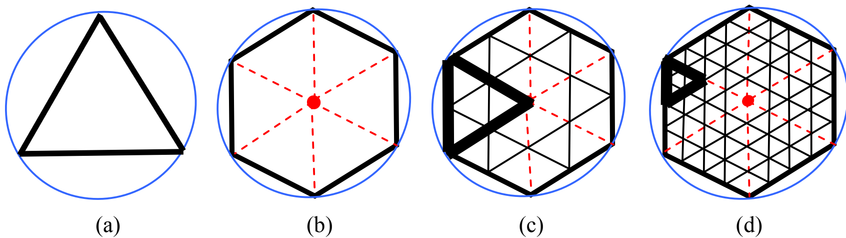


Fig. 3. First 4 levels of hierarchical Delaunay triangulation: (a) equilateral constrained Delaunay triangle, (b) hexagonal constrained Delaunay triangles, (c) 3rd level, and (d) 4th level

At this point we depart from the circle and show that there exists hierarchical constrained Delaunay structure for successive refinements of the hexagon. Later we show that the refinements remain close to the circle. We also demonstrate that a deformation of so-called "border triangles" onto the circle produces a constrained

triangulation in which border triangles are Delaunay *with respect to* non-border triangles (i.e. no non-border triangle's vertices are interior to a border triangle's circumcircle), and all non-border triangles are Delaunay.

To create the refinement, for each triangle in the current refinement, connect midpoints of each triangle edge, as shown in Fig. 3(c). Each triangle therefore generates 4 equilateral subtriangles, each of perfect quality. (Note that for an arbitrary level i (> 2) of this hierarchical refinement, there are $6 * (4^{i-2})$ triangles.) Fig. 3(d) depicts a fourth level of refinement. One triangle from the third level of refinement has thick edges to indicate its relationship to its subtriangles. We claim that, at each level of this hierarchical refinement, all triangles are Delaunay.

Theorem 1: Each level i (> 2) of the hierarchical refinement is a Delaunay triangulation of the circular pad's hexagon.

Proof: In order to establish this, we must show that the interior of each triangle's circumcircle is empty of triangle vertices. Let A be a triangle at an arbitrary level i (> 2) of this hierarchical refinement, and let $C(A)$ be A's circumcircle. Let the *star* [3] of a vertex of A be the union of all equilateral triangles in refinement i of which it is a vertex. Note that due to the way triangles are subdivided, each star's outer boundary (link [3]) is a hexagon (except for triangles touching the border of the hexagon). We define the *star union* of a triangle as the union of the stars of its vertices, and a *star vertex* is a vertex either inside or on the boundary of the triangle's star union. All triangle vertices of the refinement outside of A's star union are further away from $C(A)$ than its star vertices. Thus, if we show that none of A's star vertices are inside $C(A)$, then A is Delaunay. The vertices of A are on the boundary of $C(A)$, so they are not interior to $C(A)$. Suppose that the radius of $C(A)$ is 1 unit in length. Then each edge of A is of length $\sqrt{3}$ and the altitude of A is 3/2. The distance of the vertices of A's star union (excluding its own vertices) that are the closest to $C(A)$'s center is greater than 3/2. Thus, all these vertices of A's star union are further than 1 unit away from the circle's center and hence are clearly outside the circle. We conclude that triangle A is Delaunay. Since A was chosen arbitrarily, the entire triangulation is Delaunay. And, since the level (> 2) is also arbitrary, each triangulation level greater than 2 in the hierarchy is also Delaunay, which completes the proof. ■

Note that, in this type of triangulation, the link of each vertex that is not on a constrained segment is a regular hexagon; the vertex has degree 6. This will become relevant to Section 5, where we consider the type of triangulation generated by CGAL's 2D constrained Delaunay triangulation.

Since our primary interest is in a constrained Delaunay triangulation which preserves line segments approximating the boundary of the original pad, we now consider the maximum distance of the circle's boundary from the outer boundary of a level of the hierarchical Delaunay triangulation. At the first level, if we assume that the radius is one unit, then the maximum distance is 1/2. At all levels beyond the first, the maximum distance is $1-(\sqrt{3}/2)$.

Now consider the following method of converting a level i (> 2) in the Delaunay hierarchical triangulation to a constrained (not necessarily Delaunay) triangulation via a deformation. For this we define the *border* of the Delaunay hierarchical triangulation to be the set of triangles touching either the circle itself or the constrained edges of the level 2 hexagon. For each vertex of the border that is not already on the circle P, project

the vertex outwards onto P in a direction orthogonal to the associated constrained hexagon edge (see Fig. 4). As there are $6 * (2^{i-1} - 1)$ moving vertices, this creates a refinement of the circle containing $6 * (2^{i-1})$ constrained segments. So, the number of constrained segments doubles with each successive level of approximation.

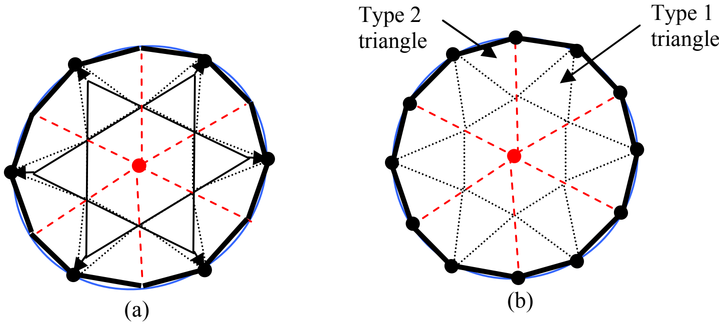


Fig. 4. Deformation of hierarchical Delaunay triangulation via orthogonal projection. (a) Deformation process. (b) Final result of deformation, with two types of border triangles (see triangle type definitions below).

Is this constrained deformed triangulation also Delaunay, regardless of its hierarchical level? Not necessarily, but we can offer some observations. We first claim that all non-border triangles remain Delaunay. This is because the number of triangle vertices is not increased by the deformation, and each moved vertex moves further *away* from every non-border triangle. Thus, no translated vertex can be interior to a non-border triangle's circumcircle. The only problematic region is therefore the deformed border triangles.

Each border triangle is one of two types: 1) triangle containing no constrained hexagon edge, and 2) triangle containing a constrained hexagon edge. (Note that there are no triangles with all 3 vertices on the circle.) Let us first consider the type 1 border triangle. It is originally an equilateral triangle which becomes isosceles under the deformation of its vertex that touches the constrained hexagon edge. Increasing movement away from the circle's center and the triangle's base, orthogonal to a hexagon edge, creates a family of isosceles triangles and associated circumcircles, with the circumcircle centers moving away from the original circle's center in the same direction as the vertex. Each successive circumcircle goes through the two base vertices and becomes closer to the base edge as the vertex moves outwards (see Fig. 5(a)). Thus, it cannot contain any vertices of non-boundary triangles. This type of triangle is therefore Delaunay *with respect to* the non-boundary triangles' vertices.

We would like to show that type 2 triangles are also Delaunay *with respect to* the non-boundary triangles' vertices. For this we refer to Fig. 5(b) and Fig. 6. In the symmetric case (Fig. 5(b)) in which both vertices move the same amount and inverted isosceles triangles are created, the circumcircles are all tangent to (and on the side opposite from) the line through the base vertex, so that the circumcircles cannot contain any non-border vertices. The extreme case occurs at a triangle containing a

hexagon vertex (see Fig. 6). In the unachievable worst case, the type 2 triangle is a right triangle. The two vertices of the triangle opposite to the right angle form the diameter of the new triangle's circumcircle. The center of this circumcircle is shown as a star in Fig. 6. While it may be possible for this circumcircle to exit the border triangle region from below, it is not possible for it to include any non-border triangle vertices because the circle is bounded on one side by its tangent line *s* (see Fig. 6).

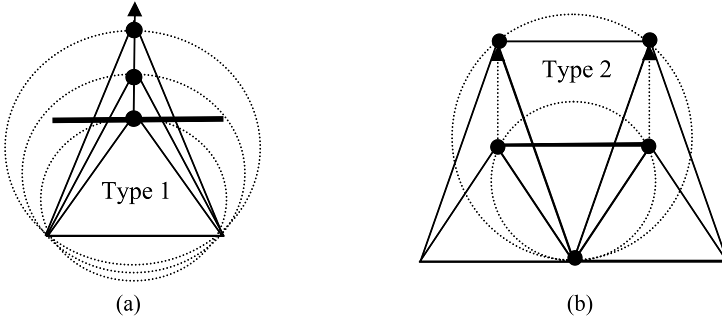


Fig. 5. Deformation of border triangles (a) Type 1 selected set of isosceles triangles generated from equilateral border triangle (bold horizontal edge is constrained hexagon edge), and (b) Type 2 in the symmetric case of equal movement of vertices

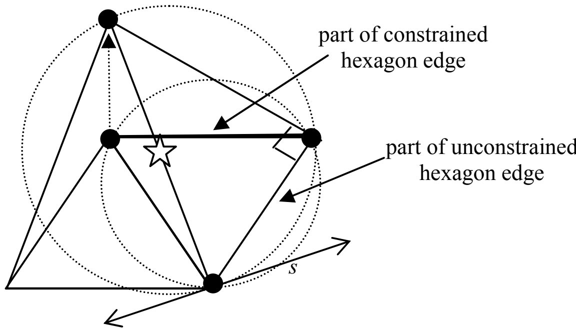


Fig. 6. Extreme deformation of type 2 border triangle: one vertex is a constrained hexagon edge endpoint

We therefore have the following result:

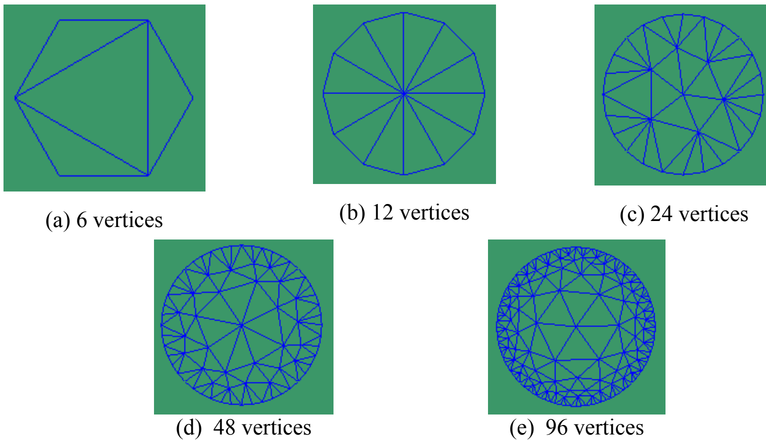
Theorem 2: Each border triangle of a hierarchical constrained Delaunay triangulation, when projected onto circle *P* orthogonally to the associated constrained hexagon edge, is Delaunay *with respect to* the non-border triangles.

Theorem 2 provides us with a constrained triangulation that possesses some Delaunay characteristics. (Note that we do not provide any Delaunay guarantee about the relationship between adjacent border triangles.) Theorem 2 applies to the $6 * (2^{i-1} - 1)$ border triangles of the $6 * (4^{i-2})$ triangles at level i (> 2).

Finally, we consider the question of whether the deformed triangulation we have created is a *constrained Delaunay* triangulation according to the definition given in Section 3. In our case, all the constrained edges of the triangulation are on the outer border (on the circle), so they cannot contribute any "blocking" of visibility. In this case, the constrained Delaunay question reduces to the actual Delaunay question. Again, we have no issue with non-border triangles, as they are Delaunay. Future work may investigate whether or not all border triangles are provably Delaunay. (See Section 5, where CGAL creates constrained triangulations that are also Delaunay.)

5 Mesh Size Trend for Pad Refinement

Here we examine how CGAL's 2D constrained Delaunay triangulation algorithm [2], which adds vertices incrementally, behaves when it performs successive refinements of an elliptical pad with a circle as a special case. During each CGAL iteration, a constrained edge is added to the current configuration by first removing edge(s) intersecting the constrained edge, and then retriangulating the holes created by the new edge. The constrained Delaunay property is then restored via edge flips [17].



(a) 6 vertices (b) 12 vertices (c) 24 vertices (d) 48 vertices (e) 96 vertices

Fig. 7. CGAL's constrained Delaunay triangulation for circular pad refinement, with number of vertices on the circle

For this experiment we use the following CGAL parameter settings: approximately 20.6 degrees for the minimal angle and 2mm for the upper bound on the length of the longest edge. Our test cases vary the elliptical aspect ratio from 1 (circle) to 2. Fig. 7 shows doubling of refinement levels, starting at 6 vertices on a circle and ending at 96. Table 1 accompanies Fig. 7 and provides details about CGAL's refinements. We compare this behavior to the constrained hierarchical triangulation (with deformation) of Section 4. Note that for 12 vertices CGAL is making the same type of pattern that we showed for 6 vertices in Fig. 3(b); there is one central Steiner point. Once CGAL goes beyond 12 vertices, note that CGAL creates nearly hexagonal structure, with larger triangles deeper inside the circular pad. Many of the vertices have degree close

to 6. Table 1 reveals average vertex degree close to 6 in several of the refinements. The average degree of vertices not on the circle is 7.71 for Fig. 7(c), 6.82 for Fig. 7(d), and 6.57 for Fig. 7(e). This is similar to the hexagonal structure of our constrained deformed triangulation in Section 4.

Table 1. CGAL's 2D constrained Delaunay meshing of circular pad refinement

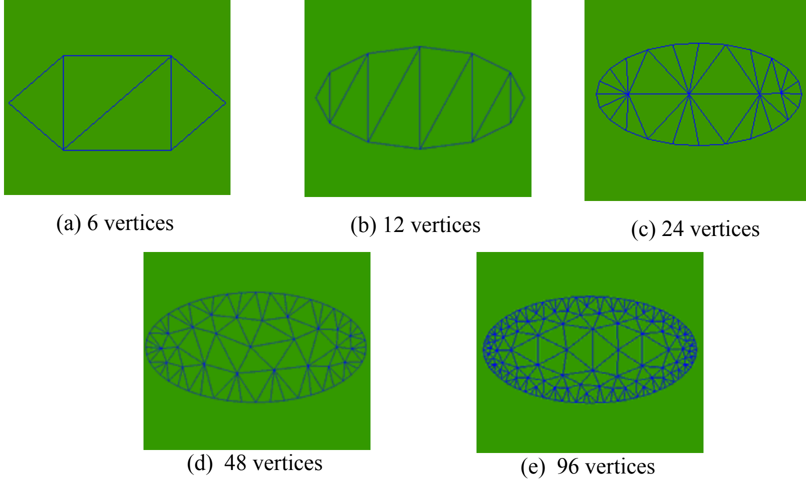
# Vertices on Circle	# Triangles	# Type 2 Border Triangles	# Steiner Points	Average Steiner Point Vertex Degree	Average Triangle Quality	Average Non-Type 2 Triangle Quality	Delaunay?
6	4	3	0	N/A	.70	1.0	Yes
12	12	12	1	12.0	.76	.76	Yes
24	36	24	7	7.71	.79	.96	Yes
48	102	48	28	6.82	.83	.94	Yes
96	244	96	77	6.57	.82	.89	Yes

Also note CGAL's creation of an outer layer of triangles that strongly resembles characteristics of the border defined in Section 4. There are Type 1 triangles which appear to be nearly isosceles. There are also Type 2 triangles. All of the border triangles can be classified as one of these two types. The number of triangles is significantly smaller than the exponential $6 * (4^{i-2})$ triangles at level i (> 2) that we derived in Section 4. In our experiments, the number of CGAL triangles increases nearly linearly as a function of the number of vertices on the circle. (In [16] we observed this linear behavior for a complete single via model embedded on a PCB with rectangular boundary. In [18], a mesh is created whose output size is linear in the number of input vertices.) Table 1 also shows the number of Type 2 border triangles in each case. On average, 66% of the triangles are of this type.

The average triangle quality is close to one in all the cases we examined. Interestingly, in our experiments all of CGAL's results are Delaunay as well as constrained Delaunay. Furthermore, average triangle quality is at least .7 for each of the 5 refinement levels. Quality is higher for non-Type 2 triangles than for Type 2 border triangles (see Table 1): average non-Type 2 triangle quality is .91. Recall that, for the hierarchical deformed triangulation of Section 4, quality is equal to 1 except for border triangles. So, this is another similarity between that triangulation and what CGAL produces. Yet another similarity can be observed by noting that the number of CGAL's Type 2 border triangles is equal to the number of constrained segments on the circular boundary. This means that no constrained segments are subdivided by the triangulation process. So, there are $6 * (2^{i-1})$ constrained segments at level i (> 2), as in the hierarchical deformed triangulation of Section 4.

For the elliptical cases, we tried aspect ratios 1.33, 1.5 and 2. Fig 8 and Table 2 show the results of aspect ratio 2. The results for aspect ratios 1.33 and 1.5 are similar to those of aspect ratio 1 (circular case described above). For aspect ratio 2, we observe that once CGAL goes beyond 24 vertices, CGAL creates nearly hexagonal structure, with larger triangles deeper inside the elliptical pad. Many of the vertices

have degree close to 6. Table 2 shows average vertex degree close to 6 in several of the refinements. The average degree of vertices not on the circle is 6.96 for Fig. 8(d), and 6.49 for Fig. 8(e). As with the circular case, again this is similar to the hexagonal structure of our constrained deformed triangulation in Section 4. Furthermore, note that in this elliptical case the mesh is Delaunay in addition to constrained Delaunay.



(a) 6 vertices (b) 12 vertices (c) 24 vertices

(d) 48 vertices (e) 96 vertices

Fig. 8. CGAL's constrained Delaunay triangulation for elliptical pad refinement with aspect ratio 2, with number of vertices on the circle

Table 2. CGAL's 2D constrained Delaunay meshing of elliptical pad refinement with aspect ratio 2

# Vertices on Ellipse	# Triangles	# Type 2 Border Triangles	# Steiner Points	Average Steiner Point Vertex Degree	Average Triangle Quality	Average Non-Type 2 Triangle Quality	Delaunay?
6	4	4	0	N/A	.89	N/A	Yes
12	10	10	0	N/A	.67	N/A	Yes
24	30	24	4	9.0	.76	.91	Yes
48	94	48	24	6.96	.82	.93	Yes
96	246	96	77	6.49	.84	.91	Yes

6 Conclusion and Future Work

In this paper, we introduce a hierarchical mesh for an elliptical PCB pad structure with various aspect ratios and show that it has several useful Delaunay quality characteristics. We show experimentally that CGAL's meshing of this structure at different resolutions yields patterns similar to our hierarchical mesh. In the future, we

may 1) investigate whether or not all border triangles are provably Delaunay; 2) explore the linear meshing size relationship in our application; 3) examine how hierarchical Delaunay triangulation applies in other features of PCB structures.

References

1. Botsch, M., et al.: ACM SIGGRAPH 2007 course 23: Geometric modeling based on polygonal meshes. Association for Computing Machinery (2007)
2. CGAL User and Reference Manual, `http://www.cgal.org`
3. Edelsbrunner, H.: Geometry and Topology for Mesh Generation. Cambridge University Press, Cambridge (2001)
4. Frey, P.J., George, P.-L.: Mesh Generation: application to finite elements. Oxford and HERMES Science Publishing, Paris (2000)
5. Goodman, J.E., O'Rourke, J. (eds.): Handbook of Discrete and Computational Geometry, 2nd edn. CRC Press, Boca Raton (2004)
6. Hall, S.H., Hall, G.W., McCall, J.A.: High-Speed Digital System Design: A Handbook of Interconnect Theory and Design Practices. John Wiley & Sons, Inc. / A Wiley-Interscience Publication (2000)
7. Holzbecher, E., Si, H.: Accuracy Tests for COMSOL – and Delaunay Meshes, `http://cds.comsol.com/access/dl/papers/5436/Holzbecher.pdf`
8. Hwang, C.-T., et al.: Partially Prism-Gridded FDTD Analysis for Layered Structures of Transversely Curved Boundary. IEEE Transactions of Microwave Theory and Techniques 48(3), 339–346 (2000)
9. Lee, S.: Efficient Finite Element Electromagnetic Analysis for High-Frequency/High Speed Circuits and Multiconductor Transmission Lines. Doctoral Dissertation, University of Illinois at Urbana-Champaign, Urbana Illinois (2009)
10. Ramahi, O.M.: Analysis of Conventional and Novel Delay Lines: A Numerical Study. Journal of Applied Computational Electromagnetic Society 18(3), 181–190 (2003)
11. Rodger, D., et al.: Finite Element Modelling of Thin Skin Depth Problems using Magnetic Vector Potential. IEEE Transactions on Magnetics 33(2), 1299–1301 (1997)
12. Thompson, J.F., Soni, B.K., Weatherill, N.P. (eds.): Handbook of Grid Generation. CRC Press, Boca Raton (1999)
13. Tsukerman, I.: A General Accuracy Criterion for Finite Element Approximation. IEEE Transactions on Magnetics 34(5), 1–4 (1998)
14. Tummala, R.R.: SOP: What Is It and Why? A New Microsystem-Integration Technology Paradigm-Moore's Law for System Integration of Miniaturized Covergent Systems of the New Decade. IEEE Transactions on Advanced Packaging 27(2), 241–249 (2004)
15. Ye, S., Daniels, K.: Triangle-based Prism Mesh Generation for Electromagnetic Simulations. In: Research Note for the 17th International Meshing Roundtable, Pittsburgh, Pennsylvania, October 12-15 (2008)
16. Ye, S., Daniels, K.: Triangle-based Prism Mesh Generation on Interconnect Models for Electromagnetic Simulations. In: 19th Annual Fall Workshop on Computational Geometry (sponsored by NSF), Tufts University, Medford, MA, November 13-14 (2009)
17. Yvinec, M.: Private communication regarding CGAL's 2D constrained Delaunay algorithm (November 2009)
18. Miller, G., Phillips, T., Sheehy, D.: Linear-Sized Meshes. In: Canadian Conference on Computational Geometry, Montreal, Quebec, August 13-15 (2008)

A Parallel Multi-start Search Algorithm for Dynamic Traveling Salesman Problem

Weiqi Li

University of Michigan - Flint, 303 E. Kearsley Street, Flint, MI 48501, U.S.A.
weli@umflint.edu

Abstract. This paper introduces a multi-start search approach to dynamic traveling salesman problem (DTSP). Our experimental problem is stochastic and dynamic. Our search algorithm is dynamic because it explicitly incorporates the interaction of change and search over time. The result of our experiment demonstrates the effectiveness and efficiency of the algorithm. When we use a matrix to construct the solution attractor from the set of local optima generated by the multi-start search, the attractor-based search can provide even better result.

Keywords: dynamic TSP, network and graphs, parallel computing.

1 Introduction

Many real-world optimization problems are inherently dynamic. Dynamic optimization problems (DOP) involve dynamic variables whose values change in time. The purpose of the optimization algorithm for DOPs is to continuously track and adapt to the changing problem through time and to find the currently best solution quickly [1],[2].

The simplest way to handle dynamic problems would be to restart the algorithm after a change has occurred. However, for many DOPs, it is more efficient to develop an algorithm that makes use of information gathered from search history. This information may be used to reduce the computational complexity of tracking the movement of the global optimum.

Due to their adaptive characteristics, evolutionary algorithms (EA) and ant colony optimization (ACO) approaches have been applied to DOPs in recent years [3]-[10].

EAs are generally capable of reacting to changes of an optimization problem. The main problem for EAs to solve DOPs is the convergence of population. Once converged, the EA loses the required diversity to adapt to the changing problem. The dynamic problem requires the EAs to maintain sufficient diversity for a continuous adaptation to the changes of the solution landscape. Several strategies have been developed to address this issue. Some examples of such strategies include maintaining and reintroducing diversity during the run [11]-[13], memory schemes [14], memory and diversity hybrid schemes [15],[16], and multi-population schemes [10],[17],[18].

The standard ACO algorithm can adapt to low-frequency change or small change in problem. Many researches on ACO for dynamic problems focus on how modifying the amount of pheromone when the change occurs. When changes to the problem are small, preserving pheromone information is useful, since the solution to the new

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 65–75, 2011.
© Springer-Verlag Berlin Heidelberg 2011

problem instance is likely to share moves with the old solution. Large changes in the problem cause the solution to change radically, so preserving pheromone information may be harmful, since it can mislead the algorithm initially and cause the algorithm to be stuck in a sub-optimal local point. Therefore, it is important that pheromone is altered during a change in such a way that useful pheromone information is kept while obsolete pheromone is reset. Many strategies for modifying pheromone information have been proposed. The approach used by Gambardella et al [19] was to reset all elements of the pheromone matrix to their initial values. Stützle and Hoos [20] suggested increasing the pheromone values proportionately to their difference to the maximum pheromone value. Guntsch and Middendorf [5],[21] discussed several approaches to modify the pheromone values.

The rapid development in the areas of robot control, dynamic logistic models, telecommunications and mobile computing systems, in which data flow are considered to be time-dependent, has caused an increasing interest in the DTSP [1],[22],[23]. Since Psaraftis [24] first introduced DTSP, a wide variety of algorithms have been proposed for DTSP. Kang et al. [25] provided a survey on some benchmarking algorithms for DTSP.

This paper introduces a parallel multi-start search algorithm for DTSP. This algorithm offers many advantages. It is flexible, adaptable, effective, and easy to implement. When a change in problem occurs, the approach repairs only the search trajectories. The search trajectories directly exhibit time-varying processing, allowing the algorithm to capture the dynamics of both the problem and search. Perhaps the most important feature of such an algorithm is its ability to implement parallel computing in its algorithm.

The remainder of this paper is organized as follows. Section 2 briefly describes the multi-start search in TSP and its solution attractor. Section 3 describes the setting of the DTSP under consideration. Section 4 explains the parallel multi-start search procedure for the DTSP. And the final section closes the paper.

2 Multi-start Search and Solution Attractor

Many heuristic search algorithms in practice are based on or derived from a general technique known as local search [26]. Local search algorithms iteratively explore the neighborhoods of solution trying to improving the current solution by local changes. However, the search spaces are limited by the neighborhood definition. Therefore, local search algorithms are locally convergent.

One way to overcome local optimality is to restart the search procedure from a new solution once a region has been explored [27]. Multi-start heuristics produce several solutions (usually local optima), and the best overall is the algorithm's output. Multi-start search helps to explore different areas in the solution space, and therefore the algorithm generates a wide sample of the local optima.

The common opinion about the local optima in a multi-start search is that they form a "big valley" in the search space, where the local optima occur relatively close to each other, and to the global optimum [28],[29]. If we start local search from several different initial points, after letting the search ran for a long time, we should find that these search trajectories would settle onto a small attractive region.

This small region is called a *solution attractor* for the local search process in that problem [30]. The solution attractor of local search process can be defined as a subset of the solution space that contains the end points of all local search trajectories. In other words, the solution attractor contains all locally optimal points. Since the globally optimal point is a special case of local optima, it is expected to be embodied in the solution attractor.

Fig. 1 presents the procedure for constructing the solution attractor of local search for a static TSP instance. The procedure is very straightforward: generating *M* locally optimal tour, storing them into a matrix *E* (called *hit-frequency matrix*), removing some unfavorable edges in *E*, and finally finding all tours contained in *E*.

```
1    procedure TSP_Attractor(Q)
2    begin
3      repeat
4        si = Initial_Tour();
5        sj = Local_Search(si);
6        Update(E(sj));
7      until Multistart = M;
8      E = Remove_Noise(E)
9      Exhausted_Search(E)
10   end
```

Fig. 1. The procedure for constructing solution attractor of local search in TSP

In the procedure, *Q* is a TSP instance. s_i is an initial tour generated by the function `Initial_Tour()`. The function `Local_Search()` runs a local search on s_i and output a locally optimal tour s_j. The function `Update(E)` records the edges in s_j into *E*. After *M* locally optimal tours are generated, the matrix *E* keeps the union of the edges in the set of *M* local optima. The matrix *E* can catch rich information about the attractor for the TSP instance [30].

The solution attractor constructed from a set of locally optimal tours contains some unfavorable edges (noise). The function `Remove_Noise()` is used to cluster the edges in *E* in an attempt to remove some noise. If we remove the edges that have low hit frequency, the remaining edges in *E* are the globally superior edges that constitute the core of the solution attractor.

Finally, an exhausted-enumeration process `Exhausted_Search()` searches the matrix *E* to identify all solutions in the attractor core. Because the attractor core in *E* represents a very small region in solution space and contains very limited number of solutions, the computational complexity in *E* is easily manageable for TSP. The solution attractor in *E* usually contains the globally optimal tour [30].

Our motivating hypothesis for using the matrix *E* is that useful information about the edges of globally optimal tour is typically contained in a suitable diverse collection of locally optimal tours. In the attractor construction procedure, a local search is used to find a locally optimal tour. However, a local search (1) lacks information needed to find all globally superior edges, and (2) lacks the architecture to use such information. The hit-frequency matrix *E* is a suitable data structure for providing such architecture and information. When each search trajectory reaches its locally optimal point, it leaves its "final footprint" in the matrix *E*. The matrix *E*

provides the architecture that allows individual tours interact along a common structure and generate the complex global structure.

There are other two propositions that are underlined the application of the hit-frequency matrix E in the procedure. The first proposition is that the globally superior edges are hit by a set of locally optimal tours with higher probabilities. Thus, the solution attractor is mainly formed by these globally superior edges. The second proposition is that a group of locally optimal tours together contains information about other locally optimal tours since each locally optimal tour shares some of its edges with other locally optimal tours. This concept motivates the procedure that takes advantage of the context where certain partial configuration of a solution often occurs as components of another solution. The strategy of "seeking the most promising partial configurations" can help circumvent the combinatorial explosion by manipulating only the most promising elements of the solution space.

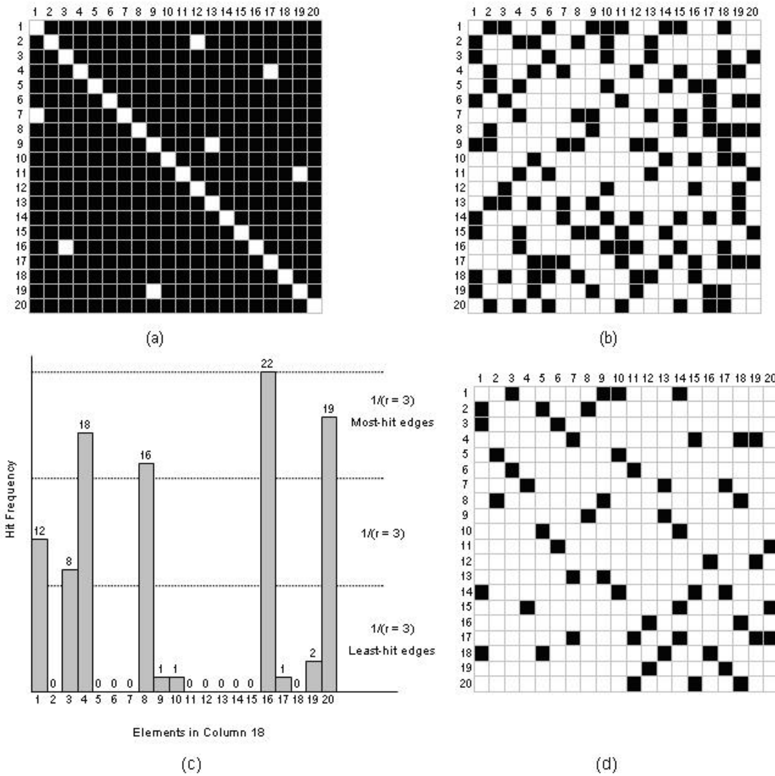


Fig. 2. An example of solution attractor construction

In fact, some researchers have utilized these properties in a set of locally optimal solutions to design their heuristic algorithms. For example, path relinking has been suggested as an approach to integrate intensification and diversification strategies in heuristic search. The path relinking approach generates new solutions by exploring

trajectories that connect high-quality solutions, by starting from one of these solutions and generating a path in the neighborhood space that leads toward the other solutions [31]-[34]. Applegate et al. [35] propose a tour-merging procedure that attempts to produce the best possible combination of tours by restricting to the edges that are present in at least one tour in the collection. Taking the union of the edges sets of the tours, they assemble the graph represented in the tours. They apply their methods on several TSPLIB instances and found the optimal tours in most trails.

Fig. 2 uses a 20-city TSP instance as an example to explain the attractor construction procedure. We started M = 100 initial tours. Because these initial tours are randomly produced, the edges should have an equal probability to be hit by these initial tours. The darkened elements in the matrix in Fig. 2(a) represent the union of the edges in these initial tours.

After applying 2-opt search technique to each of the initial tours, we obtain 100 locally optimal tours. The darkened elements in Fig. 2(b) represent the union of the edges hit by these 100 locally optimal tours. Fig. 2(c) illustrates the clustering process for the column 18 in E. We keep the most-hit cluster and remove other clusters. The darkened elements in Fig. 2(d) are the most-hit edges, representing the core of the solution attractor. We can see that, through this process, the search space is significantly reduced to a small region. Now we can use an exhausted-enumeration algorithm to find all solutions in the attractor core. In our example, the function `Exhausted_Search()` found 32 solutions in the attractor core.

3 The Dynamic Traveling Salesman Problem

DTSP is a TSP determined by a dynamic cost matrix C as follows:

$$C(t) = \{c_{ij}(t)\}_{n(t)\times n(t)} \tag{1}$$

where $c_{ij}(t)$ is the travelling cost from city i to city j at the real-world time t; $n(t)$ is the number of cities at time t. In DTSP, the number of cities can increase or decrease and the travelling costs between cities can change. The algorithmic problem has to be re-solved quickly after each change.

Many real-world dynamic problems are most naturally viewed in terms of stochastic dynamics. The dynamism implies that stochastic elements are introduced. The information on the problem is not completely known a priori, but instead is revealed to the decision maker progressively with time. Therefore, it becomes necessary to make the problem stochastic. This paper considers a random dynamic TSP. We starts with a 700-city instance with a randomly generated cost matrix C, in which each element $c(i, j) = c(j, i)$ is assigned a random integer number in the range [1, 1000]. Then the problem changes in the fashion shown in Fig. 3. The changes in the problem include three cases: Δn cities are removed, Δn cities are added, and the travelling costs for Δw edges are changed. Δn is randomly generated in the range of [10, 150] and Δw in the range of [10, 250]. After a change is made, the problem waits for Δt seconds until next change. Δt is a random number in the range of [5, 30]. The problem changes with time is in such a way that future instances are not known. In other words, the nature (dimensional and non-dimensional) of change, the magnitude of change, and the frequency of change occurs randomly in certain ranges. This setting of the problem is quite useful for routing in ad-hoc networks.

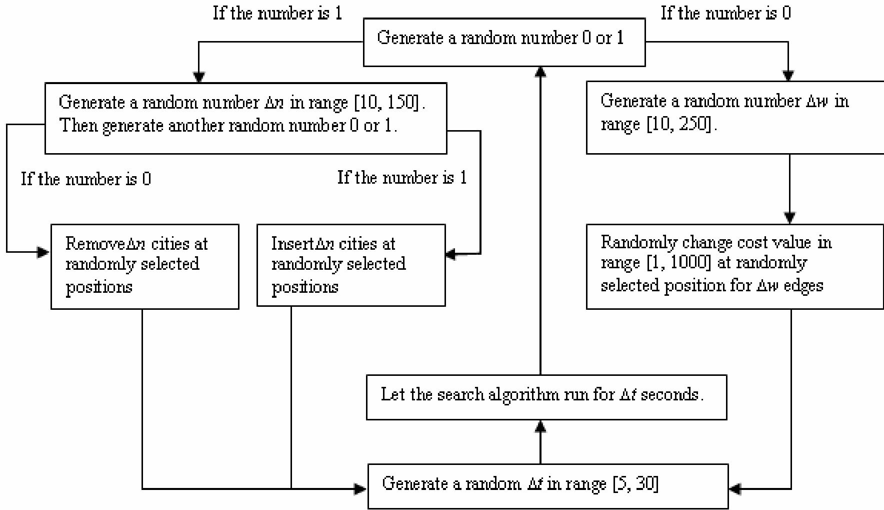


Fig. 3. Dynamic setting of the TSP

4 The Multi-start Search Procedure

When an optimization problem is in a dynamic and uncertain environment, the algorithmic challenge is to design speed-up techniques which work in a fully-dynamic scenario and do not require a time-consuming pre-processing. Fig. 4 sketches the search system for the DTSP. This system bears intrinsic parallelism in its features. Based on a common dynamic cost matrix *C*, this search system starts *M* separate search trajectories in parallel. Each of the search trajectories can store its result into a common hit-frequency matrix *E* any time during the search. A separate search processor uses an exhausted search technique to find the solutions in the matrix *E*. This system can work in real-time. Both flexibility and adaptability are built into the search system. The spread-out search trajectories can adapt to changes and keep useful search information easily. It introduces diversity without disrupting the ongoing search progress greatly.

Fig. 5 presents the attractor-based search procedure. This procedure exploits the parallelism within the algorithm. In the procedure, *C* is a dynamic cost matrix. Its size, the values in its elements, and the time of changes are all parameters subject to stochasticity within the problem described in Fig. 3. The changes in *C* include three cases: a city is removed, a city is added, and a cost element $c(i, j)$ is changed. In the first case, the study randomly selects a position (column) in *C* and removes the column and the corresponding row in *C*. In the second case, this study randomly selects a position, inserts a column and a corresponding row in *C*, and then assigns random cost values in $c(i,j)$ in the added column and row. In the third case, an element $c(i,j)$ is randomly selected and its value is replaced by a new random value.

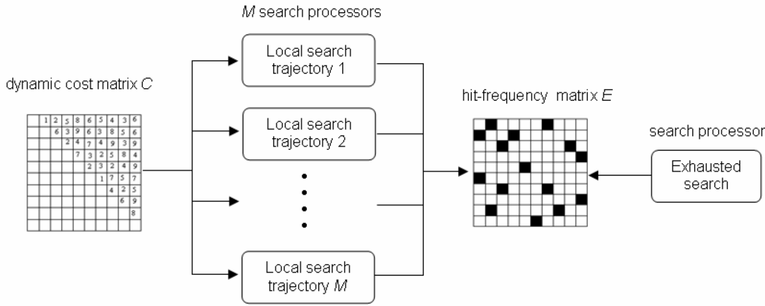


Fig. 4. Schematic structure of the search system for the DTSP

```
1    procedure DTSP_Search(C)
2    begin
3      start M separate local searches;
4      repeat
5        continue search;
6        if (Change_Flag = on)
7            (
8             repair tours in search trajectories;
9             repair matrix E;
10            )
11       if (Update_Flag = on)
12            Update(E);
14       if (Solution_Flag = on)
16            Exhausted_Search(E);
18     until Stop_Criteria
20    end
```

Fig. 5. Parallel multi-start attractor-based search procedure for DTSP

The procedure starts *M* initial points in the solution space. The *M* search trajectories are conducted in parallel. In a multi-processor system, the *M* local search can be computed separately in *M* separate processors. When a city is removed or inserted, the tours in current search trajectories will cease to be valid. We have two ways to deal with this issue. One way is to restart the *M* search trajectories for the new problem instance; another way is to repair the tours in the current search trajectories. In the case of repairing, a city is removed from a tour by connecting its respective predecessor and successor; whereas a city is inserted into a tour by connecting it with two cities in a proper position and removing the connection between those two cities.

In our procedure, each time when the size of the cost matrix *C* changes, the procedure sends a `Change_Flag` signal to all search trajectories. All search trajectories will repair their tours. The size of the matrix *E* will also be adjusted correspondingly. And then the search trajectories continue their local search based on the new cost matrix *C*. The main advantage of this method is that the algorithm keeps its diversity necessary for efficiently exploring the search space and consequently its ability to adapt to a change in the problem when such a change occurs.

The `Update_Flag` is a parameter that is used to control when we want the search trajectories to store their current tours into the matrix E. Before being updated, all elements in the matrix E are reset to zero. Since the whole information of matrix E depends only on the current solutions in the M search trajectories, it is no need to apply any particular mechanism for modifying the values in the matrix when a change in the problem occurs.

When a decision maker needs a solution for the current problem, he sends an `Information_Flag` signal to the algorithm. The algorithm triggers the Exhausted Search processor to search the matrix E and output the best solution.

Due to lack of parallel computing platform, we simulated the parallel multi-start search in a single-processor PC system (Intel Pentium processor, 1300MHz, 512mb RAM). When a change in the cost matrix C is made and a Δt is generated, we compute local search for each of M search trajectories sequentially, each spending Δt seconds. We store the tours in the M search trajectories into the matrix E. Then the solutions in E are searched by an exhausted search algorithm. The procedure then makes the next change in the cost matrix C. The procedure was implemented using MATLAB 5.

The desire to speed up the search process is particular relevant in the dynamic domain where the time to relocate the global optimum can be severely restricted by the frequency of change. The running-time constraint implies that the search is best done by using efficient local improvement heuristics like k-interchange. In our experiments, we use the 2-opt search technique in local search.

We conducted several experiments to study the search behavior of the algorithm. One experiment reported here is to study the performance of the simple multi-start search and the attractor-based search in our DTSP. We use the offline performance measure defined as follows:

$$I_t = \frac{g_t}{h_t} \tag{2}$$

where g_t is the value of optimal solution at time t, h_t is the value of the best solution found by the procedure at time t, and I_t is the index value of g_t and h_t.

We started $M = 400$ search trajectories and ran each of search trajectories for 10 seconds. Then we made the first change in the matrix C. After five changes in C were made, we start recording the search behavior of the algorithm for next 20 changes in C. Each time when a change in C was complete, we did the following steps:

1) Record the type (0 – changing values in Δw edges; 1 – removing Δn cities; 2 – inserting Δn cities) and size (Δn or Δw) of the change.
2) Use the attractor-construction procedure described in Fig. 1 to find the optimal solution for the new cost matrix C, and then calculating its value g_t.
3) If the size of the matrix C was changed, repair the tours in the M search trajectories and adjust the size of matrix E.
4) Find the best tour in the M search trajectories based on the new matrix C, calculate its value h_t, and then calculate the index value $I_{trajectory_before}$.
5) Store the tours in the M search trajectories into the matrix E, find the best tour in E, calculate its value h_t, and then calculate the index value $I_{attractor_before}$.
6) Generate the search time Δt and run each of search trajectories for Δt seconds.

7) Find the best tour in the M search trajectories, calculate its value h_t, and then calculate the index value $I_{trajectory_after}$.
8) Store the tours in the M search trajectories into the matrix E, find the best tour in E, calculate its value h_t, and then calculate the index value $I_{attractor_after}$.
9) Make the next change in the matrix C.

Table 1. Performance indexes before and after search during the 2—change period

Change Type	Change Size (n or w)	t	$I_{trajectory}$ Before	$I_{trajectory}$ After	$I_{attractor}$ Before	$I_{attractor}$ After
1	46	6	0.62	0.88	0.68	0.92
2	73	12	0.38	0.87	0.42	0.97
1	93	21	0.49	0.89	0.52	0.98
0	177	17	0.77	0.96	0.79	1
2	101	9	0.38	0.79	0.41	0.92
0	210	28	0.68	0.91	0.72	1
2	129	13	0.37	0.76	0.39	0.89
0	39	7	0.71	0.91	0.82	0.97
2	89	16	0.53	0.88	0.56	0.97
1	78	8	0.61	0.85	0.64	0.88
0	142	22	0.65	0.93	0.67	1
2	131	11	0.34	0.76	0.36	0.91
1	67	15	0.59	0.88	0.62	1
1	105	25	0.48	0.94	0.54	1
0	147	18	0.83	0.94	0.92	1
2	135	14	0.31	0.87	0.36	0.94
1	74	29	0.75	0.94	0.87	1
0	247	19	0.83	0.95	0.89	1
1	108	10	0.54	0.84	0.58	0.97
2	72		0.49	0.91	0.54	1

These steps calculate two performance indexes: I_{before} is the performance measure before search and I_{after} measures the performance after the procedure spent Δt seconds for search. The experiment compares two results: $I_{trajectory}$ is the index value for the best solution in the M search trajectories, and $I_{attractor}$ is the value for the best solution in the attractor represented by the matrix E.

Table 1 lists the collected data. The first row of the table, for example, shows that 46 cities (Δn) were removed from the problem. After repairing the search trajectories and before searching, the index for the best tour in the M search trajectories ($I_{trajectory_before}$) was 0.62 and the index for the best tour in the matrix E ($I_{attractor_before}$) was 0.68. After searching for 6 seconds (Δt), the index value for the best tour in the M search trajectories ($I_{trajectory_after}$) was improved to 0.88 and the value for the best tour in E ($I_{attractor_after}$) was improved to 0.92.

The results show that both the simple multi-start search and the attractor-based search can quickly react to the changes in the problem, and the attractor-based approach outperforms the simple multi-start search.

5 Conclusion

This paper introduces a simple and effective approach to DTSP. In our experiment, the dynamism is within the problem, the search model, and the application of the

model. Our problem is stochastic and dynamic. Our search model is dynamic because it explicitly incorporates the interaction of change and search over time, and our application is dynamic because underlying search model is repeated used as a change occurs. The result of our experiment demonstrates the effectiveness and efficiency of the algorithm. Both our benchmarking problem and attractor-based model are realistic that can be easily applied to real-world applications.

References

1. Corne, D.W., Oates, M.J., Smith, G.D.: Telecommunications Optimization: Heuristic and Adaptive Techniques. John Wiley & Sons, Chichester (2000)
2. Powell, W.B., Jaillet, P., Odoni, A.: Stochastic and Dynamic Networks and Routing. In: Ball, M.O., Magnanti, T.L., Monma, C.L., Nemhauser, G.L. (eds.) Network Routing, Handbooks in Operations Research and Management Science, vol. 8, pp. 141–296. Elsevier, Amsterdam (1995)
3. Branke, J.: Evolutionary Optimization in Dynamic Environments. Kluwer, Dordrecht (2002)
4. Eyckelhof, C.J., Snoek, M.: Ant Systems for a Dynamic TSP: Ants Caught in a Traffic Jam. In: Dorigo, M., Di Caro, G.A., Sampels, M. (eds.) Ant Algorithms 2002. LNCS, vol. 2463, pp. 88–99. Springer, Heidelberg (2002)
5. Guntsch, M., Middendorf, M.: Pheromone Modification Strategies for Ant Algorithms Applied to Dynamic TSP. In: Boers, E.J.W., Gottlieb, J., Lanzi, P.L., Smith, R.E., Cagnoni, S., Hart, E., Raidl, G.R., Tijink, H. (eds.) EvoIASP 2001, EvoWorkshops 2001, EvoFlight 2001, EvoSTIM 2001, EvoCOP 2001, and EvoLearn 2001. LNCS, vol. 2037, pp. 213–222. Springer, Heidelberg (2001)
6. Morrison, R.W.: Designing Evolutionary for Dynamic Environments. Springer, Berlin (2001)
7. Tfaili, W., Siarry, P.: A New Charged ant Colony Algorithm for Continuous Dynamic Optimization. Applied Mathematics and Computation 197, 604–613 (2008)
8. Weicker, K.: Evolutionary Algorithms and Dynamic Optimization Problems. Der Andere Verlag, Berlin (2003)
9. Yang, S., Ong, Y.-S., Jin, Y.: Evolutionary Computation in Dynamic and Uncertain Environments. Springer, Berlin (2007)
10. Younes, A., Areibi, S., Calamai, P., Basir, O.: Adapting Genetic Algorithms for Combinatorial Optimization Problems in Dynamic Environments. In: Kosinski, W. (ed.) Advances in Evolutionary Algorithms, InTech, Croatia, pp. 207–230 (2008)
11. Morrison, R.W., De Jong, K.A.: Triggered Hypermutation Revisited. In: Proceedings of 2000 Congress on Evolutionary Computation, pp. 1025–1032 (2000)
12. Tinos, R., Yang, S.: A Self-organizing Random Immigrants Genetic Algorithm for Dynamic Optimization Problems. Genetic Programming and Evolvable Machines 8(3), 255–286 (2007)
13. Wineberg, M., Oppacher, F.: Enhancing the GA's Ability to Cope with Dynamic Environments. In: Proceedings of Genetic and Evolutionary Computation Conference, GEC 2005, pp. 3–10 (2000)
14. Yang, S., Yao, X.: Population-based Incremental Learning with Associative Memory for Dynamic Environments. IEEE Transactions on Evolutionary Computation 12(5), 542–561 (2008)

15. Simões, A., Costa, E.: An Immune System-based Genetic Algorithm to Deal with Dynamic Environments: Diversity and Memory. In: Proceedings of International Conference on Neural Networks and Genetic Algorithms, pp. 168–174 (2003)
16. Yang, S.: Genetic Algorithms with Memory and Elitism Based Immigrants in Dynamic Environments. Evolutionary Computation 16(3), 385–416 (2008)
17. Branke, J., Kaussler, T., Schmidt, C., Schmeck, H.: A Multi-population Approach to Dynamic Optimization Problems. In: Proceedings of 4th International Conference on Adaptive Computing in Design and Manufacturing, pp. 299–308. Springer, Berlin (2000)
18. Ursem, R.K.: Multinational GA: Optimization Techniques in Dynamic Environments. In: Proceedings of the 2nd Genetic and Evolutionary Computation Conferences, pp. 19–26. Morgan Kaufman, San Francisco (2000)
19. Gambardella, L.-M., Taillard, E.D., Dorigo, M.: Ant Colonies for the Quadratic Assignment Problem. Journal of the Operational Research Society 50, 167–176 (1999)
20. Stützle, T., Hoos, H.: Improvements on the Ant System: Introducing MAX(MIN) Ant System. In: Proceedings of the International Conference on Artificial Neutral Networks and Genetic Algorithms, pp. 245–249. Springer, Berlin (1997)
21. Guntsch, M., Middendorf, M.: Applying Population Based ACO to Dynamic Optimization Problems. In: Dorigo, M., Di Caro, G.A., Sampels, M. (eds.) Ant Algorithms 2002. LNCS, vol. 2463, pp. 111–122. Springer, Heidelberg (2002)
22. Burkard, R.E., Deineko, V.G., Dal, R.V.: Well-solvable Special Cases of the Travelling Salesman Problem: a survey. SIAM Rev. 40(3), 496–546 (1998)
23. Li, C., Yang, M., Kang, L.: A New Approach to Solving Dynamic Travelling Salesman Problems. In: Wang, T.-D., Li, X., Chen, S.-H., Wang, X., Abbass, H.A., Iba, H., Chen, G.-L., Yao, X. (eds.) SEAL 2006. LNCS, vol. 4247, pp. 236–243. Springer, Heidelberg (2006)
24. Psaraftis, H.N.: Dynamic vehicle routing. In: Golen, B.L., Assad, A.A. (eds.) Vehicle Routing: Methods and Studies, pp. 223–248. Elsevier, Amsterdam (1988)
25. Kang, L., Zhou, A., McKay, B., Li, Y., Kang, Z.: Benchmarking Algorithms for Dynamic Travelling Salesman Problem. In: Congress on Evolutionary Computation CEC 2004, pp. 1286–1292 (2004)
26. Aarts, E., Lenstra, J.K.: Local Search in Combinatorial Optimization. Princeton University Press, Princeton (2003)
27. Martí, R., Moreno-Vega, J.M., Duarte, A.: Advanced Multi-start Methods. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics, pp. 265–281. Springer, Berlin (2010)
28. Boese, K.D., Kahng, A.B., Muddu, S.: A New Adaptive Multi-start Technique for Combinatorial Global Optimization. Oper. Res. Lett. 16, 101–113 (1994)
29. Reeves, C.R.: Landscapes, operators and heuristic search. Ann. Oper. Res. 86, 473–490 (1998)
30. Li, W.: Seeking Global Edges for Travelling Salesman Problem in Multi-start Search. J. Global Optimization. Online First Articles (2011)
31. Glover, F.: Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems. Discrete Applied Math. 65, 223–253 (1996)
32. Glover, F., Laguna, M.: Tabu Search. Kluwer, Boston (1997)
33. Laguna, M., Martí, R.: GRASP and Path Relinking for a 2-player Straight Line Crossing Minimization. INFORMS J. Comput. 11(1), 44–52 (1999)
34. Resende, M.G.C., Mart, R., Gallego, M., Duarte, A.: GRASP and Path Relinking for the Max-min Diversity Problem. Comput. And Oper. Res. 37(3), 498–508 (2010)
35. Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, Princeton (2006)

Online Dictionary Matching with Variable-Length Gaps

Tuukka Haapasalo[1], Panu Silvasti[1], Seppo Sippu[2], and Eljas Soisalon-Soininen[1]

[1] Aalto University School of Science
{thaapasa,psilvast,ess}@cs.hut.fi
[2] University of Helsinki
sippu@cs.helsinki.fi

Abstract. The string-matching problem with wildcards is considered in the context of online matching of multiple patterns. Our patterns are strings of characters in the input alphabet and of variable-length gaps, where the width of a gap may vary between two integer bounds or from an integer lower bound to infinity. Our algorithm is based on locating "keywords" of the patterns in the input text, that is, maximal substrings of the patterns that contain only input characters. Matches of prefixes of patterns are collected from the keyword matches, and when a prefix constituting a complete pattern is found, a match is reported. In collecting these partial matches we avoid locating those keyword occurrences that cannot participate in any prefix of a pattern found thus far. Our experiments show that our algorithm scales up well, when the number of patterns increases.

1 Introduction

String-pattern matching with wildcards has been considered in various contexts and for various types of wildcards in the pattern and sometimes also in the text [2–11, 13–17]. The simplest approach is to use the single-character wildcard, denoted "`.`" in `grep` patterns, to denote a character that can be used in any position of the string pattern and matches any character of the input alphabet Σ [4, 8, 15]. Generalizations of this are the various ways in which "variable-length gaps" in the patterns are allowed [2–4, 8, 11, 13, 14, 16]. Typically, a lower and upper bound is given on the number of single-character wildcards allowed between two alphabet characters in a pattern, such as "`.`$\{l,h\}$" in `grep` patterns. A special case is that any number of wildcards is allowed, called the *arbitrary-length wildcard*, denoted "`.*`" in `grep`, that matches any string in Σ^* [10, 13].

The above-mentioned solutions, except the ones by Kucherov and Rusinowitch [10] and by Zhang et al. [17], are for the single-pattern problem, that is, the text is matched against a single pattern. These algorithms [10, 17] are exceptions, because they take as input—besides the text—a set of patterns, but they are restricted to handle arbitrary-length wildcards only, and, moreover, they only find the first occurrence of any of the patterns.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 76–87, 2011.
© Springer-Verlag Berlin Heidelberg 2011

In this article we present a new algorithm that finds all occurrences of all patterns in a given pattern set, a "dictionary", in an online fashion. The patterns are strings over characters in the input alphabet Σ and over variable-length gaps, where the gaps can be specified as "`.`" (single-character wildcard), "`.`$\{l,h\}$" (gap of length l to h), or "`.*`" (gap of length 0 to ∞).

Our online algorithm performs a single left-to-right scan of the text and reports each pattern occurrence once its end position is reached, but at most one occurrence for each pattern at each character position. Each matched occurrence is identified by the pattern and its last element position in the document [2]. Because of the variable-length gaps, there can be more than one, actually an exponential number of occurrences of the same pattern at the same element position, but we avoid this possible explosion by recognizing only one occurrence in such situations.

We use the classic Aho–Corasick pattern-matching automaton (PMA) [1] constructed from the set of all keywords that appear in the patterns. A similar approach for solving the single-pattern matching problem was previously used by Pinter [15] allowing single-character wildcards in the pattern and by Bille et al. [2] allowing variable-length gaps with fixed lower and upper bounds.

Our new algorithm matches sequences of keywords that form prefixes of patterns with prescribed gaps between them. We thus record partial matches of the patterns in the form of matches of prefixes of patterns, and when a matched prefix extends up to the last keyword of the pattern, we have a true match. An important feature in our algorithm is that we use a *dynamic output function* for the PMA constructed from the keywords.

The problem definition is given Sec. 2, and our algorithm is presented in detail in Sec. 3. The complexity is analyzed in Sec. 4, based on the estimation of the number of pattern prefix occurrences in terms of the properties of the pattern set only. Experimental results, including comparisons with `grep` and `nrgrep`, are reported in Sec. 5.

2 Patterns with Gaps and Wildcards

Assume that we are given a string T of length $|T| = n$ (called the *text*) over a character alphabet Σ, whose size is assumed to be bounded, and a finite set D (called a *dictionary*) of nonempty strings (called *patterns*) P_i over characters in input alphabet Σ and over variable-length gaps. Here the *gaps* are specified as "`.`$\{l,h\}$", denoting a gap of length l to h, where l and h are natural numbers with $l \leq h$ or l is a natural number and $h = \infty$. The gap "`.{1,1}`" can also be denoted as "`.`" (the single-character wildcard or the don't-care character), and the gap "`.`$\{0,\infty\}$" as "`.*`" (the arbitrary-length wildcard).

Patterns are decomposed into keywords and gaps: the *keywords* are maximal substrings in Σ^+ of patterns. If the pattern ends at a gap, then we assume that the last keyword of the pattern is the empty string ϵ. Each pattern is considered to begin with a gap, which thus may be ϵ. For example, the pattern "`.*ab.{1,3}c.*.d..`" consists of four gaps, namely "`.*`", "`.{1,3}`", "`.*.`" (i.e., "`.`$\{1,\infty\}$"), and "`..`" (i.e., "`.{2,2}`"), and of four keywords, namely `ab`, `c`, `d`,

and ϵ. This pattern matches with, say, the input text `eeeabeecedeee`, while the pattern "`ab.{1,3}c.*.d..`" does not.

Our task is to determine all occurrences of all patterns $P_i \in D$ in text T. Like Bille et al. [2], we report a pattern occurrence by a pair of a pattern number and the character position in T of the last character of the occurrence. Because variable-length gaps are allowed, the same pattern may have many occurrences that end at the same character position; all these occurrences are reported as a single occurrence.

We number the patterns and their gaps and keywords consecutively, so that the ith pattern P_i can be represented as

$$P_i = gap(i, 1)keyword(i, 1) \ldots gap(i, m_i)keyword(i, m_i),$$

where $gap(i, j)$ denotes the jth gap and $keyword(i, j)$ denotes the jth keyword of pattern P_i.

For pattern P_i, we denote by $mingap(i, j)$ and $maxgap(i, j)$, respectively, the minimum and maximum lengths of strings in Σ^* that can be matched by $gap(i, j)$. The length of the jth keyword of pattern P_i is denoted by $length(i, j)$. We also assume that $\#keywords(i)$ gives m_i, the number of keywords in pattern P_i. For example, if the pattern "`.*ab.{1,3}c.*.d..`" is the ith pattern, we have

$\#keywords(i) = 4$,
$mingap(i, 1) = 0$, $maxgap(i, 1) = \infty$, $length(i, 1) = 2$,
$mingap(i, 2) = 1$, $maxgap(i, 2) = 3$, $length(i, 2) = 1$,
$mingap(i, 3) = 1$, $maxgap(i, 3) = \infty$, $length(i, 3) = 1$,
$mingap(i, 4) = 2$, $maxgap(i, 4) = 2$, $length(i, 4) = 0$.

3 The Matching Algorithm

For the set of all keywords in the patterns, we construct an Aho–Corasick pattern-matching automaton with a dynamically changing output function. This function is represented by sets *current-output*(q) containing *output tuples* of the form (i, j, b, e), where $q = state(keyword(i, j))$, the state reached from the initial state upon reading the jth keyword of pattern P_i, and b and e are the earliest and latest character positions in text T at which some partial match of pattern P_i up to and including the jth keyword can possibly be found. The latest possible character position e may be ∞, meaning the end of the text.

The current character position, i.e., the number of characters scanned from the input text is maintained in a global variable *character-count*. Tuples (i, j, b, e) are inserted into *current-output*(q) only at the point when the variable *character-count* has reached the value b, so that tuples (i, j, b, e) are stored and often denoted as triples (i, j, e). The function $state(keyword(i, j))$, defined from pairs (i, j) to state numbers q, is implemented as an array of $\#D$ elements, where each element is an array of $\#keywords(i)$ elements, each containing a state number.

The operating cycle of the PMA is given as Alg. 1. The procedure call *scan-next*$(character)$ returns the next character from the input text. The functions

goto and *fail* are the goto and fail functions of the standard Aho–Corasick PMA, so that $goto(state(y), a) = state(ya)$, where ya is a prefix of some keyword and a is in Σ, and that $fail(state(uv)) = state(v)$, where uv is a prefix of some keyword and v is the longest proper suffix of uv such that v is also a prefix of some keyword.

Algorithm 1. Operating cycle of the PMA with dynamic output sets

```
initialize-output()
state ← initial-state
character-count ← 0
scan-next(character)
while character was found do
   character-count ← character-count + 1
   distribute-output()
   while goto(state, character) = fail do
      state ← fail(state)
   end while
   state ← goto(state, character)
   traverse-output-path(state)
   scan-next(character)
end while
```

The function *output-fail*(q) used in the procedure *traverse-output-path* (Alg. 4) to traverse the *output path* for state q is defined by: $output\text{-}fail(q) = fail^k(q)$, where k is the greatest integer less than or equal to the length of $string(q)$ such that for all $m = 1, \ldots, k-1$, $string(fail^m(q))$ is not a keyword. Here $string(q)$ is the unique string y with $state(y) = q$, and $fail^m$ denotes the *fail* function applied m times. Thus, the output path for state q includes, besides q, those states q' in the fail path from q for which $string(q')$ is a keyword; for such states q' the dynamically changing current output can sometimes be nonempty.

The initial current output tuples, as well as all subsequently generated output tuples, are inserted through a set called *pending-output* to sets *current-output*(q). Let

$$maxdist = \max\{mingap(i,j) + length(i,j) \mid i \geq 1, j \geq 1\}.$$

The set *pending-output* is implemented as an array of *maxdist* elements such that for any character position b in the input text the element

$$pending\text{-}output(b \bmod maxdist)$$

contains an unordered set of tuples (i, j, e), called *pending output tuples*. The first pending output tuples $(i, 1, e)$, with $e = maxgap(i, 1) + length(i, 1)$, are inserted, before starting the first operating cycle, into $pending\text{-}output(b \bmod maxdist)$, where $b = mingap(i, 1) + length(i, 1)$ (see Alg. 2). At the beginning of the operating cycle, when *character-count* has reached b, all tuples (i, j, e) from the set $pending\text{-}output(b \bmod maxdist)$ are distributed into the sets *current-output*(q), $q = state(keyword(i, j)$ (see Alg. 3).

When visiting state q, the set *current-output*(q) of the PMA is checked for possible matches of keywords in the procedure call *traverse-output-path*(q) (see Alg. 4). If this set contains a tuple (i, j, e), where *character-count* $\leq e$, then a match of the jth keyword of pattern P_i is obtained. Now if the jth keyword is the last one in pattern P_i, then a match of the entire pattern P_i is obtained. Otherwise, an output tuple $(i, j+1, e')$ for the $(j+1)$st keyword of pattern P_i is inserted into the set *pending-output*$(b' \bmod maxdist)$, where

$$b' = character\text{-}count + mingap(i, j+1) + length(i, j+1), \text{ and}$$
$$e' = character\text{-}count + maxgap(i, j+1) + length(i, j+1).$$

Here $e' = \infty$ if $maxgap(i, j+1) = \infty$.

Algorithm 2. Procedure *initialize-output*()

```
for all b = 0, ..., maxdist − 1 do
   pending-output(b) ← ∅
end for
for all patterns P_i do
   b ← mingap(i, 1) + length(i, 1)
   e ← maxgap(i, 1) + length(i, 1)
   insert (i, 1, e) into the set pending-output(b mod maxdist)
end for
for all states q do
   current-output(q) ← ∅
end for
```

Algorithm 3. Procedure *distribute-output*()

```
b ← character-count
for all (i, j, e) ∈ pending-output(b mod maxdist) do
   q ← state(keyword(i, j))
   insert (i, j, e) into the list current-output(q)
end for
pending-output(b mod maxdist) ← ∅
```

The collection of the sets *current-output*(q), for states q, is implemented as an array indexed by state numbers q, where each element *current-output*(q) is an unordered doubly-linked list of elements (i, j, e), each representing a current output tuple (i, j, b, e) for some character position $b \leq$ *character-count*. The doubly-linked structure makes it easy to delete outdated elements, that is, elements with $e <$ *character-count*, and insert new elements from *pending-output*.

We also note that for each pair (i, j) (representing a single keyword occurrence in the dictionary) it is sufficient to store only one output tuple (i, j, e), namely the one with the greatest e determined thus far. To accomplish this we maintain an array of vectors, one for each pattern P_i, where the vector for P_i is indexed by j and the entry for (i, j) contains a pointer to the tuple (i, j, e) in the doubly-linked list. We assume that the insertion into *current-output*(q) in Alg. 3 first

Algorithm 4. Procedure *traverse-output-path*(*state*)

> $q \leftarrow state$
> *traversed* $\leftarrow$ false
> **while** not *traversed* **do**
>   **for** all elements (i, j, e) in the list *current-output*(q) **do**
>   **if** $e <$ *character-count* **then**
>   delete (i, j, e) from the list *current-output*(q)
>   **else if** $j = \#keywords(i)$ **then**
>   report a match of pattern P_i at position *character-count* in text T
>   **else**
> $b' \leftarrow character\text{-}count + mingap(i, j+1) + length(i, j+1)$
> $e' \leftarrow character\text{-}count + maxgap(i, j+1) + length(i, j+1)$
>   insert $(i, j+1, e')$ into *pending-output*$(b' \bmod maxdist)$
>   **end if**
>   **end for**
>   **if** $q =$ *initial-state* **then**
>   *traversed* $\leftarrow$ true
>   **else**
> $q \leftarrow$ *output-fail*(q)
>   **end if**
> **end while**

checks from this array of vectors whether or not a tuple (i, j, e') already exists, and if so, replaces e' with the greater of e and e'.

4 Complexity

The main concern in the complexity analysis is the question of how many steps are performed for each scanned input character. For each new character the procedure *traverse-output-path* (Alg. 4) is executed, and thus we need to analyze how many times the outer **while** loop and the inner **for** loop are then performed within a *traverse-output-path* call. The number of iterations of the **while** loop is the length of the output path for the current state q. The maximum length of this path is the maximum number of different keywords that all are suffixes of *string*(q) for a given state q, which implies the bound for the maximal number of performed iterations.

Additionally, within each iteration of the **while** loop the **for** loop is performed for all triples (i, j, e) that belong to *current-output*(q). Because for any pair (i, j) at most one output tuple (i, j, e) exists in *current-output*(q) for $q =$ *state*(*keyword*(i, j)) at any time, this implies that the number of iterations performed in the **for** loop for state q is bounded by the number of different occurrences of keywords equal to *keyword*$(i, j) =$ *string*(q). However, not all these occurrences have been inserted into *current-output*(q), but only those for which all preceding keyword occurrences of the pattern have been recognized.

For any two strings w_1 and w_2 composed of keywords and gaps as defined in Sec. 2, we define that w_1 is a suffix of w_2, if there are instances w_1' and w_2' of w_1 and w_2, respectively, such that w_1' is a suffix of w_2'. The instance of string w

is defined such that each gap in w is replaced by any string in Σ^* such that the gap rules are obeyed. For keyword instance (i, j) we define set $S_{i,j}$ to contain all keyword instances (i', j') where the prefix of pattern $P_{i'}$ ending with keyword instance (i', j') is a suffix of the prefix of pattern P_i ending with keyword instance (i, j).

For any two tuples (i_1, j_1, e_1) and (i_2, j_2, e_2) in *current-output*(q), either the pattern prefix ending with (i_1, j_1) is a suffix of the pattern prefix ending with (i_2, j_2), or vice versa. Thus we can conclude that the number of iterations performed in the **for** loop for q is at most $\max\{|S_{i,j}| \mid (i, j, e) \in$ *current-output*$(q)\}$. This implies further that the number of operations per input character induced by the procedure *traverse-output-path* is bounded above by the maximum size, denoted k, of the sets $S_{i,j}$, where (i, j) is any keyword instance in the dictionary D. It is clear from the matching algorithm that all other work done also has the time bound $O(kn)$, where n is the length of the input text. An upper bound for k is the number of keyword instances in the dictionary, but k is usually much less.

A better upper bound for k, instead of simply taking the number of keyword instances in D, is obtained as follows. For keyword set W denote by *pocc*(W) the number of occurrences of keywords in W in the dictionary D. Further denote by *closure*(w), for a single keyword w, the set of keywords in D that contains w and all suffixes of w that are also keywords. Then $\max\{$*pocc*$($*closure*$(w)) \mid w$ is a keyword in $D\}$ is an upper bound of k.

The preprocessing time, that is, the time spent on the construction of the PMA with its associated functions and arrays, is linear in the size of dictionary D (i.e., the sum of the sizes of the patterns in D).

In terms of the occurrences of pattern prefixes in the text it is easy to derive, for processing the text, the time complexity bound $O(Kn+$*occ*$($*pattern-prefixes*$))$, where K denotes the maximum number of suffixes of a keyword that are also keywords, and *occ*$($*pattern-prefixes*$)$ denotes the number of occurrences in the text of pattern prefixes ending with a keyword.

5 Experimental Results

We have implemented a slightly modified version of the algorithm of Sec. 3 in C++. The modifications are concerned with minor details of the organization of the current and pending output sets and with the deletion of expired output tuples. We observe that after seeing the jth keyword of pattern P_i that is followed by a gap of unlimited length, we may also consider as expired all output tuples (i, j', e) with $j' < j$. Also, we did not use the array of vectors indexed by pairs (i, j) and containing pointers to output tuples (i, j, e) (see the end of Sec. 3), but allowed the current output set for *state*$($*keyword*$(i, j))$ to contain many tuples (i, j, e).

We have run tests with a 1.2 MB input text file (the text of the book *Moby Dick* by H. Melville) using pattern files with varying number of patterns and varying number of *segments* delimited by an unlimited gap "`.*`". Let m be the length of a pattern and s the number of segments in it. We generated the patterns by taking from the input text s pieces of length m/s that are relatively

close to each other (so that the entire pattern is taken from an $8m$-character substring of the input), and by catenating these pieces together by appending a "`.*`" gap in between them. In addition, we replaced a portion of the characters in the segments with wildcards, and we converted some wildcard substrings to randomly chosen (but matching) limited variable-length gaps. Each formed pattern thus matches at least once. Partial examples of generated patterns include:

```
.omple.. .irc.* f...l.nc....n.* r.shed.to...e.* .s ..e eager
i.ot.{2,19}e.e.{0,6}.*.{0,2}cia..y.Capta.*.{1,4}ge .{2,22}eyo
```

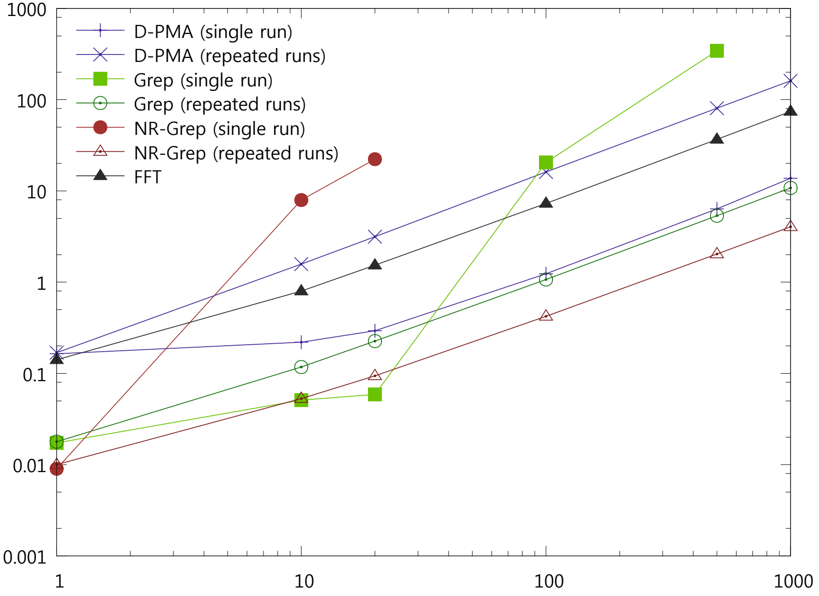
We used patterns with an average length of 80 characters when the patterns did not contain limited variable-length gaps, and 100 characters when they did (five gaps on average). The length of the variable-length gaps we used was not very high, varying on average from close to zero to around ten or twenty. (We expect the speed of our algorithm to be independent of the (minimum) length of a gap (*mingap*), while the difference of the maximum and minimum lengths of a gap does matter; *mingap*s only affect the size of the array *pending-output*.) Finally, we made 75 % of the patterns non-matching by appending a character that does not appear in the input text, at the end of the pattern; although further tests revealed that this does not affect the run time of our algorithm much. Each matching pattern usually has only one occurrence in the input text. We generated workloads with 1, 3, and 5 segments; with a total of 1, 10, 20, 100, 500, and 1000 patterns in each workload. The workloads were generated additively, so that the smaller workloads are subsets of the larger workloads.

We used the following programs in our test runs (cf. Fig. 1): (1) D-PMA, our dynamic pattern-matching algorithm; (2) FFT, the wildcard matcher based on fast Fourier transform [5] (this can only be used when the patterns do not contain arbitrary- or variable-length gaps); (3) Grep, the standard Linux command-line tool `grep`; we use the extended regular expression syntax with the `-E` parameter so that variable-length gaps can be expressed; (4) NR-Grep, the `nrgrep` Linux command-line tool by Navarro [12] (which can only handle fixed-length and arbitrary-length gaps).

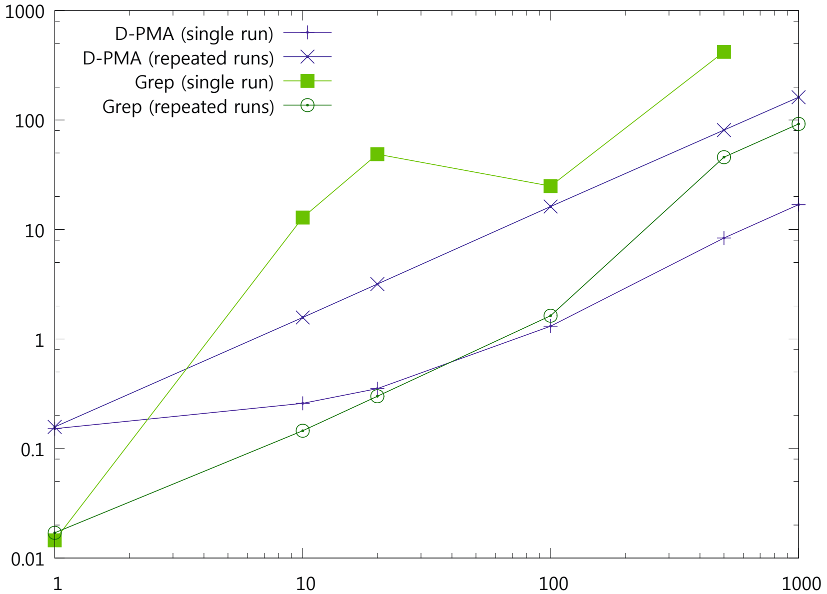
Fig. 1 shows the results of three of the test runs. The values are averages of six test runs with a standard deviation of mostly less than a percent when there are more than 20 patterns; with only a few patterns there is some small variance. The figures and further tests confirmed that `grep` performs much worse for variable-length gaps than for fixed-length gaps. On the contrary, our D-PMA algorithm has about the same performance with and without variable-length gaps.

The variants *single run* and *repeated runs* refer to how the programs were run. With *repeated runs*, each pattern of the workload was processed separately, running the program once for each pattern. This is the only way to make `grep` and `nrgrep` find occurrences individually for every pattern; in this case `grep` and `nrgrep` solve the *filtering problem* for the dictionary, that is, find the first occurrences for each pattern, if any.

With *single run*, all the patterns of a workload were fed to the program at once. With `grep`, we gave the patterns in a file with the `-f` parameter, and with `nrgrep`, we catenated the patterns together, inserting the union operator

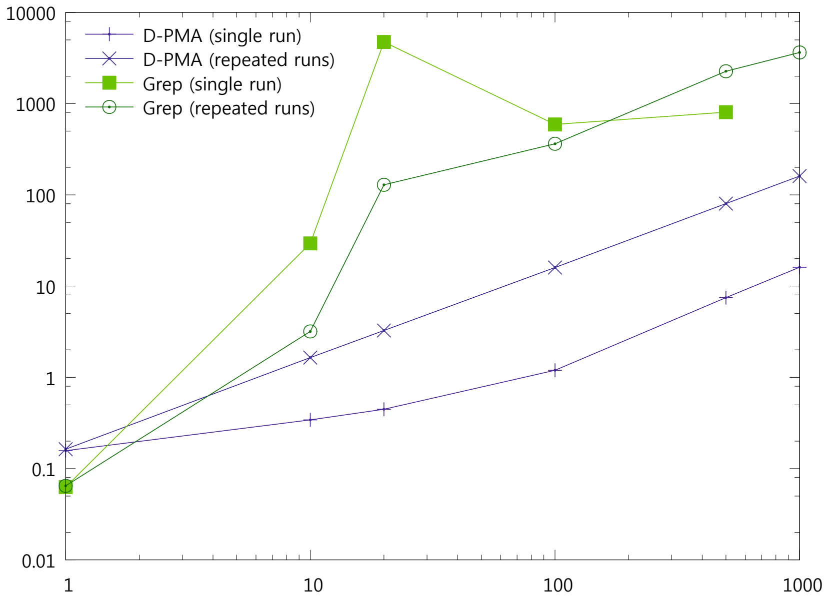


(a) Patterns with only fixed-length gaps



(b) Patterns with limited fixed- and variable-length gaps

Fig. 1. Matching times in seconds for dictionaries of increasing numbers of patterns. (NR-Grep cannot handle limited variable-length gaps).



(c) Patterns with fixed- and variable-length gaps, and four unlimited gaps

Fig. 1. *Continued*

"|" in between the pattern instances, and enclosing the patterns themselves in parentheses. In this case both `grep` and `nrgrep` only solve the *language-recognition problem* for the dictionary, that is, determine whether some pattern in the dictionary has a match; they thus stop processing the input as soon as the first match has been found. This can be seen from Figs. 1(b) and 1(c): searching for 100 patterns is faster than searching for 20 patterns, because then the first match of some pattern is found earlier.

All our tests were run on a computer with a 64-bit 2.40 GHz Intel Core 2 Quad Q6600 processor, 4 GB of main memory, and 8 MB of on-chip cache, running Fedora 14 Linux 2.6.35. The test programs were compiled with the GNU C++ compiler (`g++`) 4.5.1.

When run with a single run, both `grep` and `nrgrep` fail when there are too many patterns to process: `grep` could not complete any workload with 1000 patterns (out of memory); and `nrgrep` could not complete any workload with more than 20 patterns, but rather failed due to a possible overflow bug. Furthermore, `nrgrep` could not be run with the test workloads that included limited variable-length gaps, because `nrgrep` does not support them.

The results clearly show that our algorithm outperforms `grep` and also `nrgrep`, except when `nrgrep` was applied repeatedly (offline) for patterns with fixed-length gaps only. In that case `nrgrep` was about three times faster than our algorithm. Our algorithm scales very well to the number of patterns, for instance, for 500 patterns the online single run was ten times faster than 500 individual

runs. Moreover, we emphasize that our algorithm solves the genuine dictionary-matching problem, finding all occurrences for all the patterns, while `grep` and `nrgrep` do not. In addition, our algorithm can process multiple patterns efficiently in an online fashion, with a single pass over the input text, making it the only viable option if the input is given in a data stream that cannot be stored for reprocessing. In solving the filtering problem, our algorithm was slightly faster than when solving the dictionary-matching problem with the same pattern set and input text.

6 Conclusion

We have presented a new algorithm for string matching when patterns may contain variable-length gaps and all occurrences of a (possibly large) set of patterns are to be located. Moreover, our assumption is that pattern occurrences should be found online in a given input text, so that they are reported once their end positions have been recognized during a single scan of the text. Our solution is an extension of the Aho–Corasick algorithm [1], using the same approach as Pinter [15] or Bille et al. [2] in the sense that keywords, the maximal strings without wildcards occurring in the patterns, are matched using the Aho–Corasick pattern-matching automaton (PMA) for multiple-pattern matching.

An important feature in our algorithm is that we avoid locating keyword occurrences that at the current character position cannot take part in any complete pattern occurrence. The idea is to dynamically update the output function of the Aho–Corasick PMA. Whenever we have recognized a pattern prefix up to the end of a keyword, output tuples for the next keyword of the pattern will be inserted. In this way we get an algorithm whose complexity is not dominated by the number of all keyword occurrences in the patterns. This claim is confirmed by our experiments, which show that our algorithm outperforms `grep` and scales very well to the number of patterns in the dictionary.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. of the ACM 18, 333–340 (1975)
2. Bille, P., Li Gørtz, I., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 385–394. Springer, Heidelberg (2010)
3. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proc. of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2010), pp. 1297–1308 (2010)
4. Chen, G., Wu, X., Zhu, X., Arslan, A.N., He, Y.: Efficient string matching with wildcards and length constraints. Knowl. Inf. Syst. 10, 399–419 (2006)
5. Clifford, P., Clifford, R.: Simple deterministic wildcard matching. Inform. Process. Letters 101, 53–54 (2007)

6. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proc. of the 36th Annual ACM Symposium on Theory of Computing, pp. 90–100 (2004)
7. Fischer, M., Paterson, M.: String matching and other products. In: Proc. of the 7th SIAM-AMS Complexity of Computation, pp. 113–125 (1974)
8. He, D., Wu, X., Zhu, X.: SAIL-APPROX: an efficient on-line algorithm for approximate pattern matching with wildcards and length constraints. In: Proc. of the IEEE Internat. Conf. on Bioinformatics and Biomedicine, BIBM 2007, pp. 151–158 (2007)
9. Kalai, A.: Efficient pattern-matching with don't cares. In: Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms, pp. 655–656 (2002)
10. Kucherov, G., Rusinowitch, M.: Matching a set of strings with variable length don't cares. Theor. Comput. Sci. 178, 129–154 (1997)
11. Morgante, M., Policriti, A., Vitacolonna, N., Zuccolo, A.: Structured motifs search. J. Comput. Biol. 12, 1065–1082 (2005)
12. Navarro, G.: NR-grep: a fast and flexible pattern-matching tool. Soft. Pract. Exper. 31, 1265–1312 (2001)
13. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings. Cambridge University Press, Cambridge (2002)
14. Navarro, G., Raffinot, M.: Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. J. Comput. Biol. 10, 903–923 (2003)
15. Pinter, R.Y.: Efficient string matching. Combinatorial Algorithms on Words. NATO Advanced Science Institute Series F: Computer and System Sciences, vol. 12, pp. 11–29 (1985)
16. Rahman, M.S., Iliopoulos, C.S., Lee, I., Mohamed, M., Smyth, W.F.: Finding patterns with variable length gaps or don't cares. In: Chen, D.Z., Lee, D.T. (eds.) COCOON 2006. LNCS, vol. 4112, pp. 146–155. Springer, Heidelberg (2006)
17. Zhang, M., Zhang, Y., Hu, L.: A faster algorithm for matching a set of patterns with variable length don't cares. Inform. Process. Letters 110, 216–220 (2010)

Dynamic Arc-Flags in Road Networks

Gianlorenzo D'Angelo, Daniele Frigioni, and Camillo Vitale

Department of Electrical and Information Engineering, University of L'Aquila,
Via Gronchi, 18, I–67100, L'Aquila, Italy
{gianlorenzo.dangelo,daniele.frigioni}@univaq.it,
camillo.vitale@gmail.com

Abstract. In this work we introduce a new data structure, named *Road-Signs*, which allows us to efficiently update the Arc-Flags of a graph in a dynamic scenario. Road-Signs can be used to compute Arc-Flags, can be efficiently updated and do not require large space consumption for many real-world graphs like, e.g., graphs arising from road networks. In detail, we define an algorithm to preprocess Road-Signs and an algorithm to update them each time that a weight increase operation occurs on an edge of the network. We also experimentally analyze the proposed algorithms in real-world road networks showing that they yields a significant speed-up in the updating phase of Arc-Flags, at the cost of a very small space and time overhead in the preprocessing phase.

1 Introduction

Great research efforts have been done over the last decade to accelerate Dijkstra's algorithm on typical instances of transportation networks, such as road or railway networks (see [3] and [4] for recent overviews). This is motivated by the fact that transportation networks tend in general to be huge yielding unsustainable times to compute shortest paths. These research efforts have lead to the development of a number of so called speed-up techniques, whose aim is to compute additional data in a preprocessing phase in order to accelerate the shortest paths queries during an on-line phase. However, most of the speed-up techniques developed in the literature do not work well in dynamic scenarios, when edge weights changes occur to the network due to traffic jams or delays of trains. In other words, the correctness of these speed-up techniques relies on the fact that the network does not change between two queries. Unfortunately, such situations arise frequently in practice. In order to keep the shortest paths queries correct, the preprocessed data needs to be updated. The easiest way is to recompute the preprocessed data from scratch after each change to the network. This is in general infeasible since even the fastest methods need too much time.

Related Works. Geometric Containers [17], was the first technique studied in a dynamic scenario [18]. The key idea is to allow suboptimal containers after a few updates. However, this approach yields quite a loss in query performance. The same holds for the dynamic variant of Arc-Flags proposed in [1], where,

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 88–99, 2011.
© Springer-Verlag Berlin Heidelberg 2011

after a number of updates, the query performances get worse yielding only a low speed-up over Dijkstra's algorithm. In [15], ideas from highway hierarchies [14] and overlay graphs [16] are combined yielding very good query times in dynamic road networks. In [2], a theoretical approach to correctly update overlay graphs has been proposed, but the proposed algorithms have not been shown to have good practical performances in real-world networks. The ALT algorithm, introduced in [8] works considerably well in dynamic scenarios where edge weights can increase their value that is, when delays or traffic jams increase travel times. Also in this case, query performances get worse if too many edges weights change [5]. Summarizing, all above techniques work in a dynamic scenario as long as the number of updates is small. As soon as the number of updates is greater than a certain value, it is better to repeat the preprocessing from scratch.

Contribution. In this paper we introduce a new data structure, named *Road-Signs*, which allows us to efficiently update the Arc-Flags of a graph in a dynamic scenario. Road-Signs can be used to compute Arc-Flags, they can be efficiently updated and do not require large space consumption for many real-world graphs like, e.g., graphs arising from road networks. In detail, we define an algorithm to preprocess Road-Signs and an algorithm to update them each time that a weight increase operation occurs on an edge of the graph. As the updating algorithm is able to correctly update Arc-Flags, there is no loss in query performance. To our knowledge, the only dynamic technique known in the literature with no loss in query performance is that in [15].

We experimentally analyze the proposed algorithms in real-world road networks showing that, in comparison to the recomputation from-scratch of Arc-Flags, they yield a significant speed-up in the updating phase of Arc-Flags, at the cost of a little space and time overhead in the preprocessing phase. In detail, we experimentally show that our algorithm updates the Arc-Flags at least 62 times faster than the recomputation from scratch in average, considering the graph where the new algorithm performs worse. Moreover it performs better when the network is big, hence it can be effectively used in real-world scenarios. In order to compute and store the Road-Signs, we need an overhead in the preprocessing phase and in the space occupancy. However, we experimentally show that such an overhead is very small compared to the speed-up gained in the updating phase. In fact, considering the graph where the new algorithm performs worse, the preprocessing requires about 2.45 and 2.88 times the time and the space required by Arc-Flags, respectively.

2 Preliminaries

A road network is modelled by a weighted directed graph $G = (V, E, w)$, called *road graph*, where nodes in V represent road crossings, edges in E represent road segments between two crossings and the weight function $w : E \rightarrow \mathbb{R}^+$ represents an estimate of the travel time needed for traversing road segments. Given G, we denote as $\bar{G} = (V, \bar{E})$ the *reverse graph* of G where $\bar{E} = \{(v, u) \mid (u, v) \in E\}$. A *minimal travel time route* between two crossings S and T in a road network

corresponds to a *shortest path* from the node s representing S and the node t representing T in the corresponding road graph. The total weight of a shortest path between nodes s and t is called *distance* and it is denoted as $d(s,t)$. A partition of V is a family $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of subsets of V called *regions*, such that each node $v \in V$ is contained in exactly one region. Given $v \in R_k$, v is a *boundary node* of R_k if there exists an edge $(u,v) \in E$ such that $u \notin R_k$.

Minimal routes in road networks can be computed by shortest paths algorithm such as Dijkstra's algorithm [6]. In order to perform an s-t query, the algorithm grows a shortest path tree starting from the source node s and greedily visits the graph. The algorithm stops as soon as it visits the target node t. A simple variation of Dijkstra's algorithm is *bidirectional Dijkstra* which grows two shortest path trees starting from both nodes s and t. In detail, the algorithm performs a visit of G starting from s and a visit of $\bar{G}$ starting from t. The algorithm stops as soon the two visits meet at some node in the graph.

A widely used approach to speed up the computation of shortest paths is *Arc-Flags* [9,11], which consists of two phases: a preprocessing phase which is performed off-line and a query phase which is performed on-line. The preprocessing phase of Arc-Flags first computes a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V and then associates a *label* to each edge (u,v) in E. A label contains, for each region $R_k \in \mathcal{R}$, a *flag* $A_k(u,v)$ which is true if and only if a shortest path in G towards a node in R_k starts with (u,v). The set of flags of an edge (u,v) is called *Arc-Flags* label of (u,v). The preprocessing phase associates also Arc-Flags labels to edges in the reverse graph $\bar{G}$. The query phase consists of a modified version of bidirectional Dijkstra's algorithm: the forward search only considers those edges for which the flag of the target node's region is true, while the backward search only follows those edges having a true flag for the source node's region. The main advantage of Arc-Flags is its easy query algorithm combined with an excellent query performance. However, preprocessing is very time-consuming. This is due to the fact that the preprocessing phase grows a full shortest path tree from each boundary node of each region yielding a huge preprocessing time. This results in a practical inapplicability of Arc-Flags in dynamic scenarios where, in order to keep correctness of queries, the preprocessing phase has to be performed from scratch after each edge weight modification.

3 Dynamic Arc-Flags

Given a road graph $G = (V, E, w)$ and a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V in regions, we consider the problem of updating the Arc-Flags of G in a dynamic scenario where a sequence of weight-increase operations $C = (c_1, c_2, \ldots, c_h)$ occur on G. We denote as $G_i = (V, E, w_i)$ the graph obtained after i weight increase operations, $0 \leq i \leq h$, $G_0 \equiv G$. Each operation c_i increases the weight of one edge $e_i = (x_i, y_i)$ of an amount $\gamma_i > 0$, i.e. $w_i(e_i) = w_{i-1}(e_i) + \gamma_i$ and $w_i(e) = w_{i-1}(e)$, for each edge $e \neq e_i$ in E.

Since Arc-Flags of G are computed by considering shortest paths trees rooted at each boundary node induced by $\mathcal{R}$, a possible approach for dynamic Arc-Flags

is to maintain these trees by using e.g. the dynamic algorithm in [7]. As the number of boundary nodes in large graphs is high, this approach is impractical.

In what follows, for sake of simplicity, we consider only Arc-Flags on the graph G as the inferred properties do not change for the reverse graph $\bar{G}$. Moreover, we assume that there exists a unique shortest path for any pair of nodes in G. The extension of the data structure and algorithms to the case of multiple shortest paths is straightforward as it is enough to break ties arbitrarily during the preprocessing and updating phases. The experimental study given in the next section considers such extension.

This section is organized as follows. First, we introduce the new data structure, which we call *Road-Signs* (denoted as S) and we show how to compute Road-Signs during the preprocessing phase of Arc-Flags. Then, we give an algorithm that uses Road-Signs in order to update the Arc-Flags. Finally, as Road-Signs result to be space expensive, we give a method to store them in a compact way, by obtaining a technique which is efficient for any kind of sparse graphs as, for instance, the road graphs used in the experimental study of the next section.

Data structure. Given an edge $(u, v) \in E$ and a region $R_k \in \mathcal{R}$, the Road-Sign $S_k(u, v)$ of (u, v) to R_k is the subset of boundary nodes b of R_k, such that there exists a shortest path from u to b that contains (u, v). The Road-Signs of (u, v) are represented as a boolean vector, whose size is the overall number of boundary nodes in the network, where the i-th element is true if the i-th boundary node is contained in $S_k(u, v)$, for some region R_k. Hence, such a data structure requires $O(|E| \cdot |B|)$ memory, where B is the set of boundary nodes of G induced by $\mathcal{R}$.

The Road-Signs of G can be easily computed by using the preprocessing phase of Arc-Flags, which builds a shortest path tree from each boundary node on $\bar{G}$. Given an edge (u, v) and a region R_k, $A_k(u, v)$ is set to true if and only if (u, v) is an edge in at least one of the shortest path trees grown for the boundary nodes of R_k. Therefore, such a procedure can be easily generalized to compute also Road-Signs. In fact, it is enough to add the boundary node b to $S_k(u, v)$ if (u, v) is an edge in the tree grown for b.

Updating algorithm. Our algorithm to update Arc-Flags is based on the following Proposition, which gives us a straightforward method to compute the Arc-Flags of a graph given the Road-Signs of that graph.

Proposition 1. *Given $G = (V, E, w)$, a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V, an edge $(u, v) \in E$ and a region $R_k \in \mathcal{R}$, the following conditions hold:*

- *if $u, v \in R_k$, then $A_k(u, v) = true$;*
- *if $S_k(u, v) \neq \emptyset$, then $A_k(u, v) = true$;*
- *if u or v is not in R_k and $S_k(u, v) = \emptyset$, then $A_k(u, v) = false$.*

In what follows, we hence give an algorithm to update Road-Signs. Let us consider a weight increase operation c_i on edge (x_i, y_i). The algorithm, denoted as DynamicRoadSigns, is based on the fact that if the shortest paths from a node u to a region R_k do not contain the edge (x_i, y_i), then the Road-Signs

Phase 1: DetectAffectedNodes(G_{i-1}, c_i, R_k)
Input : Graph G_{i-1}, operation c_i on edge (x_i, y_i) and region $R_k \in \mathcal{R}$
Output: Sets $B_k(u)$, for each $u \in V$

```
1  foreach u ∈ V do
2      B_k(u) := ∅;
3  B_k(x_i) := S_k(x_i, y_i);
4  Q.push(x_i, y_i);
5  repeat
6      (u, v) = Q.pop();
7      B_old := B_k(u);
8      B_k(u) := B_k(u) ∪ (B_k(v) ∩ S_k(u, v));
9      if B_k(u) \ B_old ≠ ∅ then
10         foreach z ∈ V such that (z, u) ∈ E do
11             Q.push(z, u);
12 until Q ≠ ∅;
```

Fig. 1. First phase of algorithm DynamicRoadSigns

to R_k of the edges outgoing from u do not change as a consequence of c_i. Therefore, DynamicRoadSignsworks in two phases: the first phase, named DetectAffectedNodes, detects the set of nodes u such that a shortest path from u to b changes as a consequence of c_i (i.e. a shortest path from u to b contains edge (x_i, y_i)), where b is a boundary node in some region R_k s. t. $u \notin R_k$; the second phase, named UpdateRoadSigns, updates $S_k(u, v)$ for each region R_k and edge (u, v) where u is one of the nodes detected in the first phase.

DetectAffectedNodes consists of a modified breadth first search of the reverse graph $\bar{G}$, for each region R_k, which starts from node x_i and prunes when a node with no shortest paths to region R_k containing (x_i, y_i) is extracted. In this search, a node can be visited at most once for each boundary node of R_k. The output of this phase is a set $B_k(u)$, for each region $R_k \in \mathcal{R}$ and for each node $u \in V$, which contains the boundary nodes b of region R_k such that a shortest path from u to b contains edge (x_i, y_i). Note that, only edges (u, v) such that $B_k(u) \neq \emptyset$ for some region $R_k \in \mathcal{R}$ could change some of their Road-Signs and Arc-Flags towards region R_k, while edges (u, v) such that $B_k(u) = \emptyset$ for each $R_k \in \mathcal{R}$ do not change neither their Road-Signs nor their Arc-Flags. The pseudo-code of DetectAffectedNodes for a region $R_k \in \mathcal{R}$ is given in Fig. 1, where Q is the queue of the modified breadth first search. Operation $Q.push(x, y)$ inserts node x into Q and stores also the predecessor y of x in the visit. Operation $Q.pop()$ extracts a pair (x, y) where x is a node and y is the predecessor of x in the visit at the time when x is pushed into Q. At lines 1–3, $B_k(u)$ is initialized as $S_k(x_i, y_i)$ for $u = x_i$ and as the empty set for any other node. At lines 4–12, the graph search of $\bar{G}$ is performed, starting from node x_i. When a node u is extracted for the first time from Q, $B_k(u)$ is set to $B_k(v) \cap S_k(u, v)$ at line 8, where v is the predecessor of u in the visit at the time when u is pushed into Q. If a node u is extracted more than once from

```
Phase 2: UpdateRoadSigns(G_{i-1}, c_i, R_k, B_k)
Input  : Graph G_{i-1}, modification c_i on edge (x_i, y_i), region R_k ∈ \mathcal{R}, and sets
         B_k(u), for each u ∈ V
Output: Updated Road-Signs
1  foreach b ∈ S_k(x_i, y_i) do
2      BinaryHeap.Clear();
3      foreach u : b ∈ B_k(u) do
4          D[u, b] := ∞;
5          foreach v such that (u, v) ∈ E and b ∉ B_k(v)  do
6              Compute the distance from v to b and store it in D[v, b];
7          D[u, b] := min{w(u, v) + D[v, b] | (u, v) ∈ E and b ∉ B_k(v)};
8          if D[u, b] ≠ ∞ then
9              find the node z such that (u, z) ∈ E and b ∈ S_k(u, z);
10             S_k(u, z) := S_k(u, z) \ {b};
11             z' := argmin{w(u, v) + D[v, b] | (u, v) ∈ E and b ∉ B_k(v)};
12             S_k(u, z') := S_k(u, z') ∪ {b};
13         BinaryHeap.Push(u, D[u, b]);
14     while BinaryHeap ≠ ∅ do
15         (v, D[v, b]) :=BinaryHeap.Pop_Min();
16         foreach u such that (u, v) ∈ E and b ∈ B_k(u)  do
17             if w(u, v) + D[v, b] < D[u, b] then
18                 D[u, b] := D[v, b] + w(u, v);
19                 BinaryHeap.node(u).Decrease(u, D[u, b]);
20                 find the node z such that (u, z) ∈ E and b ∈ S_k(u, z);
21                 S_k(u, z) := S_k(u, z) \ {b};
22                 S_k(u, v) := S_k(u, v) ∪ {b};
```

Fig. 2. Second phase of algorithm DynamicRoadSigns

Q (that is, if u reaches R_k using different paths for different boundary nodes of R_k), $B_k(u)$ is updated to $B_k(u) \cup (B_k(v) \cap S_k(u,v))$ at line 8 Finally, only nodes z such that $(z,u) \in E$ and some boundary nodes have been added to $B_k(u)$ at line 8 (i.e. $B_k(v) \cap S_k(u,v) \neq \emptyset$) are inserted in Q (lines 9–11). In this way a boundary node b of region R_k is inserted in $B_k(u)$ if and only if b is contained in all the Road-Signs in some path from u to x_i in G and hence, if and only if there exists a shortest path from u to b containing (x_i, y_i).

In the second phase, UpdateRoadSigns computes the shortest paths from a node u such that $B_k(u) \neq \emptyset$ to any boundary node in $B_k(u)$, for a given region $R_k \in \mathcal{R}$, and it updates the Road-Signs accordingly. Such shortest paths are computed as follows. First, for each node u such that $b \in B_k(u)$, for a certain boundary node $b \in S_k(x_i, y_i)$, a shortest path from u to b passing only through neighbors of u whose shortest path to b do not contain (x_i, y_i), i.e. only nodes v such that $(u,v) \in E$ and $b \notin B_k(v)$, are considered. Then, the paths passing through the remaining neighbors of u are considered. The pseudo-code of

UpdateRoadSigns is given in Fig. 2. The procedure uses a binary heap which is filled during the first computation of shortest paths (Lines 3–13) and it is used during the second computation (Lines 14–22) to extract the nodes in a greedy order, mimicking Dijkstra's algorithm. The cycle at Lines 1–22 considers only boundary nodes b belonging to the Road-Sign of edge (x_i, y_i). In the cycle at lines 3–13 the shortest paths from u to b through nodes v such that $(u, v) \in E$, $b \in B_k(u)$ and $b \notin B_k(v)$, are considered. In detail, at lines 5–6 the shortest paths from each node v to b are computed and the distances are stored in a data structure called $D[v, b]$. Note that, this step can be done by using Arc-Flags. At line 7 the estimated distance $D[u, b]$ from u to b is computed. At lines 9–12 the Road-Signs are updated according to the new distance: first (line 9) the node z such that $(u, z) \in E$ and $b \in S_k(u, z)$ is found (note that there is only a single node satisfying this condition as we are assuming that there is only one shortest path for each pair of nodes); then (line 10) b is removed from the Road-Sign of (u, z) and it is added to the Road-Sign of (u, z') (line 12), where z' is the neighbor of u giving the new estimated distance (line 11). Finally, at line 13, node u is pushed in the binary heap with priority given by the computed estimated distance. At Lines 14–22 the shortest paths from u to b through nodes v such that $(u, v) \in E$, $b \in B_k(u)$, and $b \in B_k(v)$, are considered. In detail, nodes v are extracted at line 15 in a greedy order, based on the distance to b. Then, for each node u such that $(u, v) \in E$ and $b \in B_k(u)$ (lines 16–22) a relaxation step is performed at lines 17–18, followed by a decrease operation in the binary heap (line 19) and the related update of the Road-Signs at lines 20–22. Each time that the Road-Signs are updated, the related Arc-Flags are updated according to Proposition 1. In detail, given an update on $R_k(u, v)$ for certain region $R_k \in \mathcal{R}$ and edge (u, v), then $A_k(u, v)$ is set to *true* if $u, v \in R_k$ or $S_k(u, v) \neq \emptyset$, and it is set to $false$ otherwise. For simplicity, this step is not reported in the pseudo-code and it is indeed performed at lines 10, 12, 21, and 22 of UpdateRoadSigns.

Algorithm DynamicRoadSigns consists in calling procedures DetectAffectedNodes and UpdateRoadSigns, for each region $R_k \in \mathcal{R}$. The next theorem states the correctness of DynamicRoadSigns. Due to space limitations, the proof is given in the full paper.

Theorem 1. *Given $G = (V, E, w)$ and a partition $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$ of V, for each $(u, v) \in E$ and $R_k \in \mathcal{R}$,* DynamicRoadSigns *correctly updates $S_k(u, v)$ and $A_k(u, v)$ after a weight increase operation on an edge of G.*

Compacting Road Signs. Storing Road-Signs is very space consuming. Here, we give a simple method to reduce the memory space needed to store data structure S. Given a region R_k and a node $u \notin R_k$, let us denote as $B(R_k)$ the set of boundary nodes of R_k. By the definition of Road-Signs and the assumption that there exists only one shortest path between u and any boundary node b, the following two observation hold: (i) $B(R_k) = \bigcup_{(u,v)\in E} S_k(u, v)$; (ii) $S_k(u, v_1) \cap S_k(u, v_2) = \emptyset$, for each $v_1 \neq v_2$ such that $(u, v_1) \in E$ and $(u, v_2) \in E$. It follows that we can derive the Road-Sign of an edge (u, v), for an arbitrary v by the Road-Signs of other edges $(u, v') \in E$, $v' \neq v$, as $S_k(u, v) = B(R_k) \setminus$

$\bigcup_{(u,v')\in E, v'\neq v} S_k(u,v')$. In this way, we do not store the Road-Sign of edge (u,v) and we simply compute it when it is needed, by using the above formula. As we can apply this method for each node $u \in V$, we avoid to store $|V|$ Road-Signs and hence the compacted data structure requires $O((|E|-|V|)\cdot|B|)$ space, where Road-Signs are represented as $|E|-|V|$ bit-vectors. Since in sparse graphs, like e.g. road networks $|E| \approx |V|$ the space requirement of Road-Signs is very small, as it is experimentally confirmed in the next section.

4 Experimental Study

In this section, we first compare the performances of DynamicRoadSigns against the recomputation from scratch of Arc-Flags. Then, we analyze the preprocessing performances by comparing the time and space required to compute Arc-Flags against the time and space required to compute Arc-Flags and Road-Signs. The best query performances for Arc-Flags are achieved when partitions are computed by using arc-separator algorithms [12]. In this paper we used arc-separators obtained by the METIS library [10] and the implementation of Arc-Flags of [1].

Our experiments are performed on a workstation equipped with a 2.66 GHz processor (Intel Core2 Duo E6700 Box) and 8Gb of main memory. The program has been compiled with GNU g++ compiler 4.3.5 under Linux (Kernel 2.6.36).

We consider two road graphs available from PTV [13] representing the Netherlands and Luxembourg road networks, denoted as NED and LUX, respectively. In each graph, edges are classified into four categories according to their speed limits: motorways (MOT), national roads (NAT), regional roads (REG) and urban streets (URB). The main characteristics of the graphs are reported in Table 1. Due to the space requirements of Arc-Flags, we were unable to perform experiments on bigger networks.

Evaluation of the updating phase. To evaluate the performances of DynamicRoadSigns, we execute, for each graph considered and for each road category, random sequences of 50 weight-increase operations. That is, given a graph and a road category, we perform 50 weight-increase operations on edges belonging to the given category. The weight-increase amount for each operation is chosen uniformly at random in $[600, 1200]$, i.e., between 10 and 20 minutes. As performance indicator, we choose the time used by the algorithm to complete a single update during the execution of a sequence. We measure as speed-up

Table 1. Tested road graphs. The first column indicates the graph; the second and the third columns show the number of nodes and edges in the graph, respectively; the last four columns show the percentage of edges into categories: motorways (MOT), national roads (NAT), regional roads (REG), and urban streets (URB).

graph	n. of nodes	n. of edges	%MOT	%NAT	%REG	%URB
NED	892 027	2 278 824	0.4	0.6	5.1	93.9
LUX	30 647	75 576	0.6	1.9	14.8	82.7

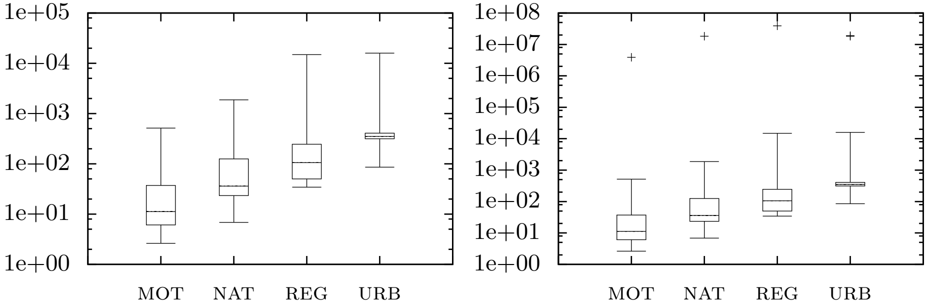


Fig. 3. Speed-up factors for the road network of the Netherlands, without (left) and with (right) outliers. For each road category, we represent minimum value, first quartile, medial value, third quartile, and maximum value.

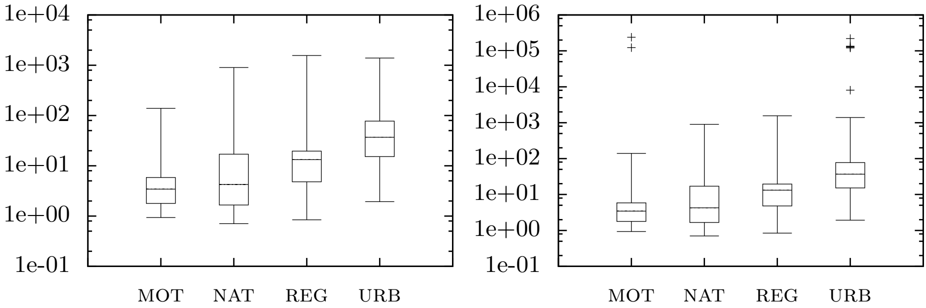


Fig. 4. Speed-up factors for the road network of Luxembourg. Without (left) and with (right) outliers. For each road category, we represent minimum value, first quartile, medial value, third quartile, and maximum value.

factor the ratio between the time required by the recomputation from scratch of Arc-Flags and that required by DynamicRoadSigns. The results are reported in Fig. 3, Fig. 4, and Table 2.

Fig. 3 shows two box-plot diagrams representing the values of the speed-up factors obtained for the road network of Netherlands, for each road category. In detail, the diagram on the left side does not represent outlier values while the diagram on the right side do. These outlier values occur when DynamicRoadSigns performs much better than Arc-Flags because the number of Road-Signs changed is very small. Here, we consider a test as outlier if the overall number of boundary nodes involved in the computation is less than 15 i.e. $|\cup_{u \in V, R_k \in \mathcal{R}} B_k(u)| \leq 15$. Even without considering outliers, the speed-up gained by DynamicRoadSigns is high in most of the cases, reaching the value of 10 000 in some cases. It is worth noting that it reaches the highest values when update operations occur on urban edges while it is smaller when they occur on motorway edges. This is due to the fact that, when an update operation occurs on urban edges, the number of shortest paths that change as a consequence of such operation is small compared to the case that an update operation occurs on motorways edges. This implies that

Table 2. Average update times and speed-up factors. The first column indicates the graph; the second column indicates the road category where the weight changes occur; the third and fourth columns show the average computational time in seconds for Arc-Flags and for DynamicRoadSigns, respectively; the fifth column shows the ratio between the values reported in the third and the fourth columns, that is the ratio of average computational times; the last column shows the average speed-up factor of DynamicRoadSigns against Arc-Flags, that is the average ratio between the computational times.

graph	cat.	avg. time Arc-Flags		avg. time DynamicRoadSigns		ratio		avg. speed-up	
NED	MOT	2 418.09	2 413.99	246.73	92.82	9.80	25.99	51.30	425.32
	NAT	2 397.14		74.71		32.08		169.82	
	REG	2 420.72		27.91		86.73		470.48	
	URB	2 416.22		7.63		316.67		1053.03	
LUX	MOT	8.25	8.28	2.96	2.04	2.79	4.06	11.70	62.87
	NAT	8.24		3.05		2.70		47.07	
	REG	8.32		1.46		5.70		78.06	
	URB	8.32		0.54		15.41		119.39	

Table 3. Preprocessing time. The first column shows the graph; the second one shows the number of regions; the third one shows the preprocessing time required for computing only Arc-Flags; the fourth column shows the preprocessing time required for computing both Arc-Flags and Road-Signs; and the last column shows the ratio between the values reported in the fourth and the third column.

graph	n. of regions	prep. time AF (sec.)	prep. time AF + RS (sec.)	ratio
NED	128	2 455.21	4 934.10	2.01
LUX	64	8.29	20.33	2.45

DynamicRoadSigns, which selects the nodes that change such shortest paths and focus the computation only on such nodes, performs better than the recomputation from-scratch of the shortest paths from any boundary node. Fig. 4 is similar to Fig. 3 but it is referred to the road network of Luxembourg. The properties highlighted for NED hold also for LUX. We note that, for NED, the speed-up factors achieved are higher than that achieved for LUX. This can be explained by the different sizes of the networks. In fact, when an edge update operation occurs, it affects only a part of the graph, hence only a subset of the edges in the graph need to update their Arc-Flags or Road-Signs. In most of the cases this part is small compared to the size of the network and, with high probability, it corresponds to the subnetwork close to the edge increased or closely linked to it. In other words, it is unlike that a traffic jam in a certain part of the network affects the shortest paths of another part which is far or not linked to the first one. Clearly, this fact is more evident when the road network is big and this explains the different performances between NED and LUX. Moreover, this allows us to state that DynamicRoadSigns would perform better if applied in networks bigger than those used in this paper, as continental networks.

Table 4. Preprocessing space requirements. The first column shows the graph; the second one shows the number of regions; the third one shows the space required for storing Arc-Flags; the fourth one shows space required for storing both Arc-Flags and Road-Signs by using the compact storage; and the last column shows the ratio between the values reported in the fourth and the third column.

graph	n. of regions	space AF (B)	space AF and RS (B)	ratio
NED	128	36 461 184	64 612 836	1.77
LUX	64	604 608	1 744 531	2.88

As a further measure of the performances of DynamicRoadSigns against the recomputation from-scratch of Arc-Flags, we report the *average* computational time and speed-up factors in Table 2. It is evident here that DynamicRoadSigns outperforms the recomputation from-scratch by far and that it requires reasonable computational time which makes Road-Signs a technique suitable to be used in practice.

Evaluation of the preprocessing phase. Regarding the preprocessing phase, in Tables 3 and 4 we report the computational time and the space occupancy required by Arc-Flags and DynamicRoadSigns. Table 3 shows that, for computing Road-Signs along with Arc-Flags, we need about 2 times the computational time required for computing only Arc-Flags, which is a very small overhead compared to the speed-up gained in the updating phase. The same observation can be done regarding the space occupancy. In fact, Table 4 shows that the space required for storing both Road-Signs and Arc-Flags is between 1.77 and 2.88 that required to store only Arc-Flags. It is worth noting that without the compact storage of data structure S described in the previous section, S would require 12.78 and 4.13 times more space for NED and LUX, respectively.

5 Conclusions

We proposed a technique to correctly update Arc-Flags in dynamic graphs. In particular, we introduced the Road-Sign data structure, which can be used to compute Arc-Flags, can be efficiently updated and does not require large space consumption. Therefore, we gave two algorithms to compute the Road-Signs in the preprocessing phase and to update them each time that a weight increasing occurs. We experimentally analyzed the proposed algorithms and data structures in road networks showing that they yields a significant speed-up in the updating phase, at the cost of a small space and time overhead in the preprocessing phase.

The proposed algorithms are able to cope only with weight increase operations which is the most important case in road networks where the main goal is to handle traffic jams. However, when a weight decrease operation occurs (e.g. when a the traffic jams is over) a recomputation from scratch is needed. Therefore, an interesting open problem is to find efficient algorithms to update Road-Signs after weight decrease operations.

References

1. Berrettini, E., D'Angelo, G., Delling, D.: Arc-flags in dynamic graphs. In: 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS 2009). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Germany (2009)
2. Bruera, F., Cicerone, S., D'Angelo, G., Stefano, G.D., Frigioni, D.: Dynamic multi-level overlay graphs for shortest paths. Mathematics in Computer Science 1(4), 709–736 (2008)
3. Delling, D., Hoffmann, R., Kandyba, M., Schulze, A.: Chapter 9. Case Studies. In: Müller-Hannemann, M., Schirra, S. (eds.) Algorithm Engineering. LNCS, vol. 5971, pp. 389–445. Springer, Heidelberg (2010)
4. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In: Lerner, J., Wagner, D., Zweig, K.A. (eds.) Algorithmics of Large and Complex Networks. LNCS, vol. 5515, pp. 117–139. Springer, Heidelberg (2009)
5. Delling, D., Wagner, D.: Landmark-Based Routing in Dynamic Graphs. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 52–65. Springer, Heidelberg (2007)
6. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
7. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully dynamic algorithms for maintaining shortest paths trees. Journal of Algorithms 34(2), 251–281 (2000)
8. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A* Search Meets Graph Theory. In: 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA 2005), pp. 156–165 (2005)
9. Hilger, M., Köhler, E., Möhring, R.H., Schilling, H.: Fast Point-to-Point Shortest Path Computations with Arc-Flags. In: Shortest Path Computations: Ninth DIMACS Challenge. DIMACS Book, vol. 24 (2009)
10. Karypis, G.: METIS - A Family of Multilevel Partitioning Algorithms (2007)
11. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths. Static Networks with Geographical Background 22, 219–230 (2004)
12. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning Graphs to Speedup Dijkstra's Algorithm. ACM J. Exp. Algorithmics 11, 2.8 (2006)
13. PTV AG - Planung Transport Verkehr (2008), http://www.ptv.de
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. Sanders, P., Schultes, D.: Dynamic Highway-Node Routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
16. Schulz, F., Wagner, D., Zaroliagis, C.: Using Multi-Level Graphs for Timetable Information in Railway Systems. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409, pp. 43–59. Springer, Heidelberg (2002)
17. 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)
18. Wagner, D., Willhalm, T., Zaroliagis, C.: Geometric Containers for Efficient Shortest-Path Computation. ACM J. Exp. Algorithmics 10, 1.3 (2005)

Efficient Routing in Road Networks with Turn Costs

Robert Geisberger and Christian Vetter

Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
{geisberger,christian.vetter}@kit.edu

Abstract. We present an efficient algorithm for shortest path computation in road networks with turn costs. Each junction is modeled as a node, and each road segment as an edge in a weighted graph. Turn costs are stored in tables that are assigned to nodes. By reusing turn cost tables for identical junctions, we improve the space efficiency. Preprocessing based on an augmented node contraction allows fast shortest path queries. Compared to an edge-based graph, we reduce preprocessing time by a factor of 3.4 and space by a factor of 2.4 without change in query time.

Keywords: route planning, banned turn, turn cost, algorithm engineering.

1 Introduction

Route planning in road networks is usually solved by computing shortest paths in a suitably modeled graph. Each edge of the graph has an assigned weight representing, for example, the travel time. There exists a plethora of speed-up techniques to compute shortest paths in such weighted graphs [1]. The most simple model maps junctions to nodes and road segments to edges. However, this model does not consider turn costs. Turn costs are important to create a more realistic cost model and to respect banned turns.

To incorporate turn costs, usually a pseudo-dual of the simple model is used [2,3], modeling road segments as nodes and turns between two consecutive road segments as edges. Thus the edges in the simple model become nodes in the pseudo-dual. Therefore, we will refer to the result of the simple model as *node-based graph* and to the pseudo-dual as *edge-based graph*. The advantage of the edge-based graph is that no changes to the speed-up techniques are required to compute shortest paths, as only edges carry a weight. The drawback is a significant blowup in the number of nodes compared to the *node-based graph*. To avoid this blowup, we will extend the node-based graph by assigning *turn cost tables* to the nodes, i.e., junctions, and show how to efficiently perform precomputation for a major speed-up technique. We further reduce the space consumption by identifying junctions that can share the same turn cost table.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 100–111, 2011.
© Springer-Verlag Berlin Heidelberg 2011

1.1 Related Work

There is only little work on speed-up techniques with respect to turn costs. The main reason is that incorporating them is seen as redundant due to the usage of an edge-based graph [2,3].

Speed-up techniques for routing in road networks can be divided into hierarchical approaches, goal-directed approaches and combinations of both. Delling et al. [1] provide a recent overview of them. In this paper, we focus on the technique of *node contraction* [4,5,6]. It is used by the most successful speed-up techniques known today [6,7,8]. The idea is to remove unimportant nodes and to add shortcut edges (*shortcuts*) to preserve shortest path distances. Then, a bidirectional Dijkstra finds shortest paths, but never needs to relax edges leading to less important nodes. Contraction hierarchies (CH) [6] is the most successful hierarchical speed-up technique using node contraction; it contracts in principle one node at a time. Node contraction can be further combined with goal-directed approaches to improve the overall performance [7]. The performance of CH on edge-based graphs has been studied by Volker [9].

2 Preliminaries

We have a *graph* $G = (V, E)$ with *edge weight function* $c : E \to \mathbb{R}_+$ and *turn cost function* $c^t : E \times E \to \mathbb{R}_+ \cup \{\infty\}$. An edge $e = (v, w)$ has source node v and target node w. A turn with cost ∞ is *banned*. A path $P = \langle e_1, \ldots, e_k \rangle$ must not contain banned turns. The *weight* is $c(P) = \sum_{i=1}^{k} c(e_i) + \sum_{i=1}^{k-1} c^t(e_i, e_{i+1})$. The problem is to compute a path with smallest weight between e_1 and e_k, that is a *shortest path*. Note that source and target of the path are edges and *not* nodes.

To compute shortest paths with an algorithm that cannot handle a turn cost function, the *edge-based* graph [3] $G' = (V', E')$ is used with $V' = E$ and $E' = \{(e, e') \mid e, e' \in E$, target node of e is source node of e' and $c^t(e, e') < \infty\}$. We define the edge weight by $c' : (e, e') \mapsto c(e') + c^t(e, e')$. Note that the cost of a path $P = \langle e_1, \ldots, e_k \rangle$ in the edge-based graph misses the cost $c(e_1)$ of the first edge. Nevertheless, as each path between e_1 and e_k misses this, shortest path computations are still correct.

To compute a shortest path in the edge-based graph, any shortest path algorithm for non-negative edge weights can be used. E. g., Dijkstra's algorithm

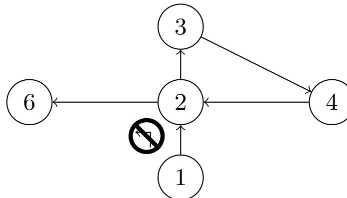


Fig. 1. Graph (unit distance) that restricts the turn $1 \to 2 \to 6$. Therefore, the shortest path $1 \to 2 \to 3 \to 4 \to 2 \to 6$ visits node 2 twice.

computes shortest paths from a single source node to all other nodes by settling nodes in non-decreasing order of shortest path distance. However, on a node-based graph with turn cost function, settling nodes is no longer sufficient, see Figure 1. Instead, we need to settle edges, i.e., nodes in the edge-based graph. Initially, for source edge e_1 the tentative distance $\delta(e_1)$ is set to $c(e_1)$, all other tentative distances are set to infinity. In each iteration the unsettled edge $e = (v, w)$ with smallest tentative distance is *settled*. Then, for each edge e' with source w, the tentative distance $\delta(e')$ is updated with $\min\big(\delta(e'), \delta(e) + c^{\mathrm{t}}(e, e') + c(e')\big)$. This resulting *edge-based Dijkstra* successfully computes shortest paths. By storing parent pointers, shortest paths can be reconstructed in the end.

3 Turn Cost Model

We take into account two kinds of turn costs: a limited turning speed inducing a turn cost and a fixed cost added under certain circumstances. Banned turns have infinite turn cost.

3.1 Fixed Cost

We add a fixed turning cost when turning left or right. In addition to this a fixed cost is applied when crossing a junction with traffic lights.

3.2 Turning Speed

Tangential acceleration. We can use the turn radius and a limit on the tangential acceleration a to compute a limit on the turning speed v: $\max_v = \sqrt{\max_a * \text{radius}}$. Given a lower limit on the resolution δ_{min} of the underlying data we estimate the resolution δ the turn is modeled with: When turning from edge e into edge e' the respective edge length are l and l'. Then, δ is estimated as the minimum of l, l' and δ_{min}. We compute the turn radius from the angle α between the edges: $\text{radius} = \tan(\alpha/2) * \delta/2$.

Traffic. When turning into less important road categories we restrict the maximum velocity to simulate the need to look out for traffic. We differentiate between left and right turns, e.g. it might not be necessary to look out for incoming traffic when turning right. Furthermore, we limit the maximum turning speed when pedestrians could cross the street.

Turn costs. We can derive turn costs from the turn speed limit $\max_v$. Consider a turn between edges e and e'. When computing $c(e)$ and $c(e')$ we assumed we could traverse these edges at full speed v and v'. When executing the turn between them, we now have to take into account the deceleration and acceleration process. While traversing edge e we assume deceleration a_{dec} from v down to $\max_v$ at the latest point possible and while traversing edge e' we assume immediate start of acceleration a_{acc} from $\max_v$ to v'. The turn cost we apply is the difference

between this time and the projected travel time on the edges without acceleration and deceleration. The resulting turn cost is $c^{\mathrm{t}}(e, e') = (v - \max_v)^2/(2 * a_{\mathrm{dec}}) + (v' - \max_v)^2/(2 * a_{\mathrm{acc}})$. Of course, this is only correct as long as the edge is long enough to fully accelerate and decelerate.

4 Node Contraction

Node contraction without turn costs was introduced in an earlier paper [6]. The basic idea behind the contraction of a node v is to add shortcuts between the neighbors of v to preserve shortest path distances. In a graph without turn costs, this is done by checking for each incoming edge (u, v) from remaining neighbor u and for each outgoing edge (v, w) to remaining neighbor w, whether the path $\langle u, v, w \rangle$ is the only shortest path between u and w. If this is the case, then a *shortcut* (u, w) representing this path needs to be added. This is usually decided using a node-based Dijkstra search (*local search*) starting from node u. The neighbors u and w are more important than node v, as they are not contracted so far. A query algorithm that reaches node u or w never needs to relax an edge to the less important node v, as the shortcut can be used. The query is bidirected and meets at the most important node of a shortest path. This shortest path $P' = \langle e_1, e_2, \ldots, e_k \rangle$ found by the query can contain shortcuts. To obtain the path P in the original graph, consisting only of *original edges*, each shortcut e' needs to be replaced by the path $\langle e'_1, \ldots, e'_\ell \rangle$ it represents.

4.1 With Turn Costs

Turn restrictions complicate node contraction. As we showed in Section 2, we need an edge-based query instead of a node-based one. Therefore, we have to preserve shortest path distances between edges, and not nodes. An important observation is that it is sufficient to preserve shortest path distances only between original edges. This can be exploited during the contraction of node v if the incoming edge (u, v) and/or the outgoing edge (v, w) is a shortcut. Assume that (u, u') is the *first* original edge of the path represented by (u, v) and (w', w) is the *last* original edge of the path represented by (v, w). We do not need to add a shortcut for $\langle (u, v), (v, w) \rangle$ if it does not represent a shortest path between (u, u') and (w', w) in the original graph. The weight of the shortcut is the sum of the weights of the original edges plus the turn costs between the original edges.

We introduce the following notation: A shortcut $(u \to u', w' \to w)$ is a shortcut between nodes u and w, the first original edge is (u, u') and the last original edge is (w', w). If two nodes are connected by an arrow, e. g., $u \to u'$, then this always represents an original edge (u, u'). A node-triplet connected by arrows, e. g., $u'' \to u \to u'$, always represents a turn between the original edges (u'', u) and (u, u') with cost $c^{\mathrm{t}}(u'' \to u \to u')$.

Local search using original edges. Now that we have established the basic idea of node contraction in the presence of turn costs, we will provide more details.

An observation is that we cannot avoid parallel edges and loops between nodes in general, if they have different first or last original edge. Therefore, we can only uniquely identify an edge by its endpoints *and* the first and last original edge. Loops at a node v make the discovery of potentially necessary shortcuts more complicated, as given an incoming edge $(u \to u', v' \to v)$ and outgoing edge $(v \to v'', w' \to w)$, the potential shortcut $(u \to u', w' \to w)$ may not represent $\langle (u \to u', v' \to v), (v \to v'', w' \to w) \rangle$ but has to include one or more loops at v in between. This can happen, e. g., in Figure 1, if nodes 2, 3 and 4 are contracted, then there has to be a shortcut between nodes 1 and 6 including a loop. Therefore, we use the local search to not only to decide on the necessity of a shortcut, but also to find them. The local search from incoming edge $(u \to u', v' \to v)$ computes tentative distances $\delta(\cdot)$ for *original* edges only. Initially, for each remaining edge $(u \to u', x' \to x)$ with first original edge $u \to u'$, $\delta(x' \to x) := c(u \to u', x' \to x)$, and all other distances are set to infinity. To settle an original edge $x' \to x$, for each edge $e' = (x \to x'', y' \to y)$ with source x, the tentative distance $\delta(y' \to y)$ is updated with $\min\big(\delta(y' \to y), \delta(x' \to x) + c^{\mathrm{t}}(x' \to x \to x'') + c(e')\big)$. A shortcut $(u \to u', w' \to w)$ is added iff the path computed to $w' \to w$ only consists of the incoming edge from u, the outgoing edge to w and zero or more loops at v in between. Otherwise a *witness* is found of smaller or equal weight. The weight of the shortcut is $\delta(w' \to w)$.

4.2 Optimizations

The algorithm described so far preserves shortest path distances between all remaining uncontracted nodes. However, as our query already fixes the first and last original edge of the shortest path, we can further reduce the number of shortcuts. It would be sufficient to only add a shortcut $(u \to u', w' \to w)$ if there are two original edges, a source edge (u'', u) and a target edge (w, w'') such that $\langle (u'', u), (u \to u', w' \to w), (w, w'') \rangle$ would be only shortest path in the remaining graph together with (u'', u) and (w, w'') but without node v. This allows to avoid a lot of unnecessary and 'unnatural' shortcuts. E. g., a query starts from a southbound edge of a highway but the journey should go north. Naturally, one would leave at the first exit, usually the target of the edge, and reenter the highway northbound. Our improvement allows to avoid shortcuts representing such changes of direction.

Aggressive local search. We can use the above observation to enhance the local search in a straightforward manner. Instead of executing a local search from the original edge (u, u'), we perform a local search separately from each original incoming edge (u'', u). Then, we check for each original edge (w, w'') whether the shortcut is necessary. While this approach can reduce the number of shortcuts, it increases the number of local searches, and therefore the preprocessing time.

Turn replacement. To avoid performing a large amount of local queries we try to combine the searches from all the original edges incoming to u into one search. We cannot start from all these original edges simultaneously while still computing

just a single distance per original edge. It is better to start from all original edges outgoing from u simultaneously. We initialize the local search as in Section 4.1. Furthermore, we consider all other remaining edges $(u \to u'_2, x' \to x)$ outgoing from u. However, as we now replace a turn $u'' \to u \to u'$ by a turn $u'' \to u \to u'_2$, an *outgoing turn replacement difference* $\overrightarrow{r}(u \to u', u'_2) := \max_{u''} \big(c^{\mathrm{t}}(u'' \to u \to u'_2) - c^{\mathrm{t}}(u'' \to u \to u')\big)$ needs to be added to account for the different turn costs, see Figure 2. Note that we consider the worst increase in the turn cost over all incoming original edges of u. So $\delta(x' \to x) := \overrightarrow{r}(u \to u', u'_2) + c(u \to u'_2, x' \to x)$. The local search settles original edges as before, but has a different criterion to add shortcuts. We add a shortcut $(u \to u', w' \to w)$ with weight $\delta(w' \to w)$ iff the path computed to $w' \to w$ only consists of the incoming edge $(u \to u', v' \to v)$, the outgoing edge to $(v'' \to v, w' \to w)$ and zero or more loops at v in between, *and* none of the other edges incoming to w offers a witness. Consider a path computed to an original edge $w'_2 \to w$ incoming to node w. If we consider this path instead of the one computed to $w' \to w$, we would replace the turn $w' \to w \to w''$ by the turn $w'_2 \to w \to w''$. A *incoming turn replacement difference* $\overleftarrow{r}(w'_2, w' \to w) := \max_{w''} \big(c^{\mathrm{t}}(w'_2 \to w \to w'') - c^{\mathrm{t}}(w' \to w \to w'')\big)$ is required to account for the different turn costs. We do not need to add a shortcut if $\overleftarrow{r}(w'_2, w' \to w) + \delta(w'_2 \to w) < \delta(w' \to w)$.

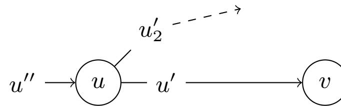


Fig. 2. If a witness uses turn $u'' \to u \to u'_2$ instead of $u'' \to u \to u'$, we have to account for the turn cost difference $c^{\mathrm{t}}(u'' \to u \to u'_2) - c^{\mathrm{t}}(u'' \to u \to u')$

Loop avoidance. Even with the turn replacement approach of the previous paragraph, there can still be a lot of unnecessary loop shortcuts. E. g., in Figure 1, assume that nodes 1 and 6 are not present. After the contraction of nodes 3 and 4, there would be a loop shortcut at node 2 although it is never necessary. We only need to add a loop shortcut, if it has smaller cost than a direct turn. That is, considering a loop shortcut $(u \to u', u'' \to u)$ at node u, if there are neighbors u'_2 and u''_2 such that $c^{\mathrm{t}}(u'_2 \to u \to u') + c(u \to u', u'' \to u) + c^{\mathrm{t}}(u'' \to u \to u''_2) < c^{\mathrm{t}}(u'_2 \to u \to u''_2)$.

Limited local searches. Without turn costs, local searches only find witnesses to avoid shortcuts. Therefore, they can be arbitrarily pruned, as long as the computed distances are upper bounds on the shortest path distances [6]. That ensures that all necessary, and maybe some superfluous shortcuts are added. But with turn costs, a local search also needs to find the necessary shortcuts. Therefore, we cannot prune the search for those. We limit local searches by the number of settled original edges. Once we settled a certain number, we only settle original edges whose path from the source is a prefix of a potential shortcut. Furthermore, if all reached but unsettled original edges cannot result in a potential shortcut, we prune the whole search.

5 Preprocessing

To support turn costs, the CH preprocessing [6] only needs to use the enhanced node contraction described in Section 4. The preprocessing performs a contraction of all nodes in a certain order. The original graph together with all shortcuts is the result of the preprocessing. The order in which the nodes are contracted is deduced from a node priority consisting of: (a) The edge quotient, i.e., the quotient between the amount of shortcuts added and the amount of edge removed from the remaining graph. (b) The original edge quotient, i.e., the quotient between the number of original edges represented by the shortcuts and the number of original edges represented by the edges removed from the remaining graph. (c) The hierarchy depth, i.e., an upper bound on the amount of hops that can be performed in the resulting hierarchy. Initially, we set $\text{depth}(u) = 0$ and when a node v is contracted, we set $\text{depth}(u) = \max(\text{depth}(u), \text{depth}(v)+1)$ for all neighbors u. We weight (a) with 8, (b) with 4 and (c) with 1 in a linear combination to compute the node priorities. Nodes with higher priority are more important and get contracted later. The nodes are contracted in parallel by computing independent node sets with a 2-neighborhood [10].

6 Query

The query computes a shortest path between two original edges, a source $s \to s'$ and a target $t' \to t$. It consists of two Dijkstra-like searches that settle original edges (cf. Section 2) one in forward direction starting at $s \to s'$, and one in backward direction starting at $t' \to t$. The only restriction is that it never relaxes edges leading to less important nodes. Both search scopes meet at the most important node z of a shortest path. E. g., the forward search computes a path to $z' \to z$, and the backward search computes a path to $z \to z''$. As usually $z' \neq z''$, when we settle an original edge $x' \to x$ in forward direction, we need to check whether the backward search reached any outgoing edge $x \to x''$, and vice versa. Such a path with smallest weight among all meeting nodes is a shortest path.

Stall-on-demand. As our search does not relax all edges, it is possible that an original edge $x' \to x$ is settled with suboptimal distance. In this case, we can prune the search at this edge, since the computed path cannot be part of a shortest path. To detect some of the suboptimally reached edges, the *stall-on-demand technique* [5] can be used, but extended to the scenario with turn costs: The edge $x' \to x$ is settled suboptimally if there is an edge $(y \to y', x' \to x)$ and an original edge $y'' \to y$ such that $\delta(y'' \to y) + c^{\mathrm{t}}(y'' \to y \to y') + c(y \to y', x' \to x) < \delta(x' \to x)$.

Path unpacking. To unpack the shortcuts into original edges, we can store the *middle node* whose contraction added the shortcut. Then, we can recursively unpack shortcuts [6]. Note that if there are loops at the middle node, they may be part of the shortcut. A local search that only relaxes original edges incident to the middle node can identify them.

7 Turn Cost Tables

We can store the turn cost function $c^{\mathrm{t}} : E \times E \rightarrow \mathbb{R}_{+} \cup \{\infty\}$ efficiently using a single table per node. Each adjacent incoming and outgoing edge gets assigned a local identifier that is used to access the table. To avoid assigning bidirectional edges two identifiers we take care to assign them the same one in both directions. This can easily be achieved by assigning the bidirectional edges the smallest identifiers.

To look up the correct turn costs in the presence of shortcuts we need to store additional information with each shortcut: A shortcut $(u \rightarrow u', w' \rightarrow w)$ has to store the identifier of (u, u') at u and the identifier of (w', w) at w.

Storing these identifiers does not generate much overhead as their value is limited by the degree of the adjacent node.

7.1 Redundancy

Since the turn cost tables model the topology of road junction they tend to be similar. In fact many tables model exactly the same set of turn costs. We can take advantage of this by replacing those instances with a single table. To further decrease the amount of tables stored we can rearrange the local identifiers of a table to match another table. Of course, we have to take care to always assign bidirectional edges the smallest identifiers.

Given a reference table t and a table t' we check whether t' can be represented by t by trying all possible permutations of identifiers. Bidirectional identifiers are only permuted amongst themselves. Because the amount of possible permutations increases exponentially with the table size we limit the number of permutations tested. Most junctions only feature a limited amount of adjacent edges and are not affected by this pruning. Nevertheless, it is necessary as the data set may contain erroneous junctions with large turn cost tables.

To avoid checking a reference table against all other tables we compute hash values $h(t)$ for each table t. $h(t)$ has the property that if $h(t) \neq h(t')$ neither t can be represented by t' nor t' by t. We compute $h(t)$ as follows: First, we sort each row of the table, then sorting the rows lexicographically. Finally, we compute a hash value from the resulting sequence of values.

We use this hash values to greedily choose an unmatched table and match as many other tables to it as possible.

8 Experiments

Environment. The experimental evaluation was done on a machine with four AMD Opteron 8350 processors (Quad-Core) clocked at 2 GHz with 64 GiB of RAM and 2 MiB of Cache running SuSE Linux 11.1 (kernel 2.6.27). The program was compiled by the GNU C++ compiler 4.3.2 using optimization level 3.

Instances. We use three road networks derived from the publicly available data of OpenStreetMap, see Table 1. Travel times where computed using the MoNav Motorcar Profile [11]. Using the node-based model with turn cost tables requires about 30% less space than the edge-based model. That is despite the fact that in the node-based model, we need more space per node and edge: Per node, we need to store an additional pointer to the turn cost table (4 Bytes), and an offset to compute a global identifier from the identifier of an original edge (u, u') local to a node u (4 Bytes). Per edge, we need to additionally store the local identifier of the first and last original edge (2×1 Byte rounded to 4 Byte due to alignment).

Table 1. Input instances. In the edge-based model, a node requires 4 Bytes (pointer in edge array), and an edge requires 8 Bytes (target node + weight + flags). In the node-based model, a node requires additional 8 Bytes (pointer to turn cost table + offset to address original edges), and an edge requires additional 4 Bytes (first and last original edge). An entry in a turn cost table requires 1 Byte.

graph	model	nodes		edges		turn cost tables		
		[$\times 10^6$]	[MiB]	[$\times 10^6$]	[MiB]	[$\times 10^3$]	%	[MiB]
Netherlands	node-based	0.8	9.4	1.9	22.2	79	9.9%	0.8
	edge-based	1.9	7.4	5.2	39.7	-	-	-
Germany	node-based	3.6	41.3	8.5	97.1	267	7.4%	3.1
	edge-based	8.5	32.4	23.1	176.3	-	-	-
Europe	node-based	15.0	171.1	35.1	401.3	834	5.6%	9.5
	edge-based	35.1	133.8	95.3	727.0	-	-	-

Redundant turn cost tables. Already for the Netherlands, only one table per ten nodes needs to be stored. The best ratio is for the largest graph, Europe, with one table per 18 nodes. This was to be expected as most unique junctions types are already present in the smaller graphs. Identifying the redundant tables is fast, even for Europe, it took only 20 seconds.

Node Contraction. Preprocessing is done in parallel on all 16 cores of our machine. We compare the node contraction in the node-based and the edge-based model in Table 2. In the node-based model, we distinguish between the *basic* contraction without the optimizations of Section 4.2, the *aggressive* contraction mentioned in Section 4.2, and the contraction using turn replacement (TR) and loop avoidance (LA). Clearly, TR+LA is the best contraction method. The basic contraction requires about a factor 3–4 times more preprocessing time, about 5–7 times more space, and results in 3–4 times slower queries. It misses a lot of witnesses which leads to denser remaining graphs, so that its preprocessing is even slower than the aggressive contraction's. The aggressive contraction finds potentially more witnesses as TR+LA, but shows no significant improvement, neither in preprocessing space nor query performance. For Europe, its performance even slightly decreases, potentially due to the limited local searches and because a different node order is computed. Furthermore, its preprocessing is about a factor 3 slower, because we execute several local searches per neighbor with an edge incoming to the contracted node.

Table 2. Performance of contraction hierarchies (TR = turn replacement, LA = loop avoidance)

graph	model	contraction	preprocessing time [s]	space [MiB]	%	query time [μs]	settled edges
Netherlands	node-based	basic	66	31.9	144%	1 177	713
		aggressive	57	7.0	32%	319	367
		TR + LA	19	7.0	32%	315	363
	edge-based	regular	63	46.6	117%	348	358
Germany	node-based	basic	250	124.2	128%	2 339	1 158
		aggressive	244	17.3	18%	735	594
		TR + LA	73	17.3	18%	737	597
	edge-based	regular	253	183.9	104%	751	535
Europe	node-based	basic	1 534	592.2	148%	4 075	1 414
		aggressive	1 318	117.4	29%	1 175	731
		TR + LA	392	116.1	29%	1 107	651
	edge-based	regular	1308	817.1	112%	1061	651

We will compare the contraction in the edge-based model only to the contraction in the node-based model using TR+LA. Its preprocessing is about 3.4 times faster than in the edge-based model. One reason is that there are about 2.3 fewer nodes need to be contracted, and TR+LA, compared to the aggressive contraction, needs only one local search per incoming edge. We argue that the additional speed-up comes from the fact that contracting junctions instead of road segments works better. Note that there is a fundamental difference in contracting a node in the node-based and edge-based model: Adding a shortcut in the node-based model would map to an additional node in the edge-based model. We observe that the total space required including preprocessed data is about a factor 2.4 larger for the edge-based model.

Furthermore, in the node-based model, bidirected road segments can be stored more efficiently by using forward/backward flags. In comparison, assume that you have a bidirected edge in the original edge-based graph. This implies that the edge represents two U-turns between (u, v) and (v, u), see Figure 3. Therefore, bidirected road segments cannot be stored efficiently in the edge-based model.

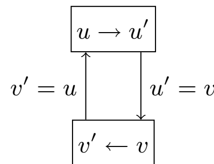


Fig. 3. A bidirected edge in the original edge-based graph between two original edges (u, u') and (v, v') in the node-based graph. Because a turn from (u, u') to (v, v') is possible, $u' = v$, and because a turn from (v, v') to (u, u') is possible, $v' = u$. Therefore, both turns are U-turns.

Query. Query performance is averaged over 10 000 shortest path distance queries run sequentially on a single core of our machine. Source and target edge have been selected uniformly at random. The resulting distances were compared to a plain edge-based Dijkstra for correctness. Interestingly, the best query times that can be achieved in both models are almost the same. One reason might be that both queries settle original edges. For the smaller graphs the query time is even a bit faster in the node-based model, because most of the turn cost tables fit into cache, thus causing almost no overhead.

9 Conclusions

Our work shows the advantages of the node-based model over the edge-based one. The node-based model stores tables containing the turn costs. By identifying redundant turn cost tables, we can decrease the space required to store them by one order of magnitude. Still, our query has to settle original edges so that we need to store a local identifier per edge and an offset to obtain a global identifier per node. Therefore, a query in the original node-based graph is the same as in the original edge-based graph, but storing the graph requires 30% less space.

Our preprocessing based on node contraction works better in the node-based model in terms of preprocessing time (factor ≈ 3.4) and space (factor ≈ 2.4) without affecting the query time. To augment the node-based contraction to turn cost tables, we had to augment the local searches to not only identify witnesses, but also shortcuts, because parallel and loop shortcuts can be necessary. To restrict the node contraction to one local search per incoming edge (factor ≈ 3 faster) without missing too many witnesses, we developed the techniques of turn replacement and loop avoidance.

9.1 Future Work

We want to integrate the turn cost tables into an existing mobile implementation of contraction hierarchies [12]. To further reduce the space requirements of the turn cost tables, we can approximate their entries. This not only reduces the number of different turn cost tables we need to store, but also the bits required to store a table entry.

Acknowledgement. The authors would like to thank Dennis Luxen for his valuable comments and suggestions.

References

1. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In: Lerner, J., Wagner, D., Zweig, K.A. (eds.) Algorithmics of Large and Complex Networks. LNCS, vol. 5515, pp. 117–139. Springer, Heidelberg (2009)
2. Caldwell, T.: On Finding Minimum Routes in a Network With Turn Penalties. Communications of the ACM 4(2) (1961)

3. Winter, S.: Modeling Costs of Turns in Route Planning. GeoInformatica 6(4), 345–361 (2002)
4. 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)
5. Schultes, D., Sanders, P.: Dynamic Highway-Node Routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
6. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
7. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. ACM Journal of Experimental Algorithmics 15(2.3), 1–31 (2010); Special Section devoted to WEA 2008
8. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 231–242. Springer, Heidelberg (2011)
9. Volker, L.: Route Planning in Road Networks with Turn Costs, Student Research Project (2008), http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/volker_sa.pdf
10. Vetter, C.: Parallel Time-Dependent Contraction Hierarchies, Student Research Project (2009), http://algo2.iti.kit.edu/download/vetter_sa.pdf.
11. Vetter, C.: MoNav (2011), http://code.google.com/p/monav/
12. Vetter, C.: Fast and Exact Mobile Navigation with OpenStreetMap Data. Master's thesis, Karlsruhe Institute of Technology (2010)

On Minimum Changeover Cost Arborescences

Giulia Galbiati[1], Stefano Gualandi[2], and Francesco Maffioli[3]

[1] Dipartimento di Informatica e Sistemistica, Università degli Studi di Pavia
[2] Dipartimento di Matematica, Università degli Studi di Pavia
{giulia.galbiati,stefano.gualandi}@unipv.it
[3] Politecnico di Milano, Dipartimento di Elettronica e Informazione
maffioli@elet.polimi.it

Abstract. We are given a digraph $G = (N, A)$, where each arc is colored with one among k given colors. We look for a spanning arborescence T of G rooted (wlog) at node 1 and having minimum changeover cost. We call this the Minimum Changeover Cost Arborescence problem. To the authors' knowledge, it is a new problem. The concept of changeover costs is similar to the one, already considered in the literature, of reload costs, but the latter depend also on the amount of commodity flowing in the arcs and through the nodes, whereas this is not the case for the changeover costs. Here, given any node $j \neq 1$, if a is the color of the single arc entering node j in arborescence T, and b is the color of an arc (if any) leaving node j, then these two arcs contribute to the total changeover cost of T by the quantity d_{ab}, an entry of a k-dimensional square matrix D. We first prove that our problem is NPO-complete and very hard to approximate. Then we present Integer Programming formulations together with a combinatorial lower bound, a greedy heuristic and an exact solution approach. Finally, we report extensive computational results and exhibit a set of challenging instances.

1 Introduction

The problem that we consider in this paper and that we call Minimum Changeover Cost Arborescence, is formulated on a digraph $G = (N, A)$ having n nodes and m arcs, where each arc comes with a color (or label) out of a set C of k colors. The problem looks for a spanning arborescence T rooted at a specific node, say node 1 (wlog), and having minimum *changeover cost*, a cost that we now describe. We assume that G contains at least one spanning arborescence having node 1 as root, i.e., a cycle-free spanning subgraph containing a directed path from node 1 to all other nodes of G. Let T be any such arborescence; obviously T has only one arc entering each node except node 1. Consider any node j of T different from the root. A cost at j is paid for each outgoing arc and depends on the colors of this arc and of the one entering j. Such costs are given via a $k * k$ matrix D of non-negative rationals: the entry $d_{a,b}$ specifies the cost to be paid at node j for one of the outgoing arcs (if any) colored b, when the incoming arc of j is colored a. We define the *changeover cost at j*, denoted by $d(j)$, as the sum of the costs at j paid for each of its outgoing arcs. We call

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 112–123, 2011.
© Springer-Verlag Berlin Heidelberg 2011

the *changeover cost of* T, denoted by $d(T)$, the sum of the changeover cost $d(j)$, over all nodes j different from 1.

Similar problems have recently received some attentions in the literature [1–5], where the entries of our matrix D are called *reload costs*. However in all these problems the objective functions depend on the amount of commodity flowing in the arcs or edges of the given graph, whereas this is not the case for the problem that we consider. In the seminal work [1] and in [2] the problem of finding a spanning tree with minimum reload cost diameter is thoroughly analyzed; in [3] the problem of finding a spanning tree which minimizes the sum of the reload costs of all paths between all pairs of nodes is discussed. The problems of minimum reload cost path-trees, tours, flows are studied in [4], whereas the minimum reload $s-t$-path, trail and walk problems are analyzed in [5].

The motivation for introducing changeover costs comes from the need to model in a real network the fixed costs for installing, in each node, devices to support the changes of carrier, modeled here with changes of color.

The Minimum Changeover Cost Arborescence (MinCCA for short) is thoroughly analyzed in this paper, both from a theoretical and a computational point of view. In Section 2 we show that it is NPO-complete and very hard to approximate. We present in Section 3 some Integer Programming formulations and in Section 4 a greedy algorithm. The last section show an exact solution approach, extensive computational results and a set of challenging instances.

2 Complexity

In this section we prove that MinCCA is NPO PB-complete and hard to approximate, even in the very restrictive formulation given in Theorem 1 below. This result is obtained by exhibiting a reduction from another NPO PB-complete problem, namely problem Minimum Ones, which is defined as follows. Given a set $Z = \{z_1, ..., z_n\}$ of n boolean variables and a collection $C = \{c_1, ..., c_m\}$ of m disjunctive clauses of at most 3 literals, the problem looks for a subset $Z' \subseteq Z$ having minimum cardinality and such that the truth assignment for Z that sets to true the variables in Z' satisfies all clauses in C. The trivial solution, returned when the collection of clauses is unsatisfiable, is set Z.

In [6] it is shown that Minimum Ones is NPO PB-complete, and in [7] that it is not approximable within $|Z|^{1-\epsilon}$ for any $\epsilon > 0$.

We now present the reduction that we will use in the proof of Theorem 1. Given any instance I of Minimum Ones we construct the corresponding instance I' of MinCCA as follows (see Figure 1 for an example). Graph $G = (N, A)$ has the vertex set N that contains a vertex $a_0 = 1$, the vertices a_i, b_i, c_i, d_i, for each $i = 1, ..., n$, and the vertices $c_1, ..., c_m$ corresponding to the elements of C. Moreover N contains, for each $i = 1, ..., n$, many couples of vertices $z_i^j, \overline{z}_i^j$, with j having values from 1 to the maximum number of times that either z_i or $\overline{z}_i$ appear in the clauses of C. The arc set A of G contains all arcs (a_i, a_{i+1}), for $i = 0, ..., n-1$, all arcs (a_i, b_i), (a_i, c_i), (b_i, d_i), (c_i, d_i), (d_i, z_i^1), $(d_i, \overline{z}_i^1)$, for $i = 1, ..., n$, and all arcs (z_i^j, z_i^{j+1}) and $(\overline{z}_i^j, \overline{z}_i^{j+1})$, for the increasing values of j,

starting from 1. Moreover there is an arc from z_i^j to $\overline{z}_i^j$ and vice versa, for each involved i and j. Finally A contains an arc from z_i^j (resp. $\overline{z}_i^j$) to vertex c_k if the j-th occurrence of variable z_i (resp. $\overline{z}_i$) in the clauses of C, if it exists, appears in clause c_k. The colors for the arcs of G are taken from the set $\{r, g, v, b\}$. Color r is assigned to all arcs (a_i, a_{i+1}), (a_i, b_i), and (a_i, c_i); color g to all arcs (b_i, d_i), (d_i, z_i^1), (z_i^j, z_i^{j+1}) and all arcs from some vertex z_i^j to some vertex c_k; symmetrically color v is assigned to all arcs (c_i, d_i), $(d_i, \overline{z}_i^1)$, $(\overline{z}_i^j, \overline{z}_i^{j+1})$ and all arcs from some vertex $\overline{z}_i^j$ to some vertex c_k; finally all arcs from z_i^j to $\overline{z}_i^j$ and vice versa are colored with color b. The matrix D of costs is defined so that $d(r, g) = 1$, $d(b, g) = d(b, v) = d(g, v) = d(v, g) = M$, with $M > n^2$ being a suitable big integer, and all other costs equal to 0. If we let n' denote the number of nodes of G, it is not difficult to see that $n' \leq \beta n^3$, for some constant β, with $\beta > 1$ and independent from the number n of boolean variables. For this purpose it is enough to convince ourselves that the following inequalities hold:

$$n' \leq 1 + 6n + 7m \leq 1 + 6n + 7\binom{2n}{3} = 1 + 6n + \frac{7}{3}(2n^2 - n)(2n - 2) \leq \beta n^3.$$

It is also not difficult to see that $opt(I) \leq opt(I')$ since to any solution of I' of cost $t < M$ there corresponds a solution to I having the same cost t; if, on the other hand, $opt(I') \geq M$ then $opt(I) \leq n < opt(I')$. Notice that the equality $opt(I) = opt(I')$ holds iff I is a satisfiable instance, since to any non trivial solution of I there corresponds a solution to I' having the same cost, whereas to a trivial solution of I there corresponds a solution to I' having a cost greater than M.

Now we can prove the following theorem.

Theorem 1. *Problem* MinCCA *is NPO-complete. Moreover, there exists a real constant* α*,* $0 < \alpha < 1$*, such that the problem is not approximable within* $\alpha\sqrt[3]{n^{1-\varepsilon}}$*, for any* $\varepsilon > 0$*, even if formulated on graphs having bounded in and out degrees, costs satisfying the triangle inequality and 4 colors on the arcs.*

Proof. We use the reduction described in this section. Obviously the graph constructed from an instance I of Minimum Ones is a graph $G = (N, A)$ that has maximum in-degree and out-degree equal to 3, costs that satisfy the triangle inequality (see [4] for a definition) and uses $k = 4$ colors. Moreover we know that the number n' of its nodes satisfies the inequality $n' \leq \beta n^3$, with $\beta > 1$.

We show now that, if we let $\alpha = \frac{1}{\sqrt[3]{\beta}}$, then MinCCA is not approximable within $\alpha\sqrt[3]{n'^{1-\varepsilon}}$, for any $\varepsilon > 0$. Suppose on the contrary that there exists an $\bar{\varepsilon} > 0$ and that MinCCA is approximable within $\alpha\sqrt[3]{n'^{1-\bar{\varepsilon}}}$. The algorithm that we now describe could then be used to approximate Minimum Ones within $n^{1-\bar{\varepsilon}}$, contrary to the result in [7].

The algorithm, given an instance I of Minimum Ones, would construct the corresponding instance I' of MinCCA and then find an approximate solution for it, i.e. a spanning arborescence T having a changeover cost $d(T)$ satisfying the inequality $\frac{d(T)}{opt(I')} \leq \alpha\sqrt[3]{n'^{1-\bar{\varepsilon}}}$. If $d(T) \geq M$ the algorithm would return the

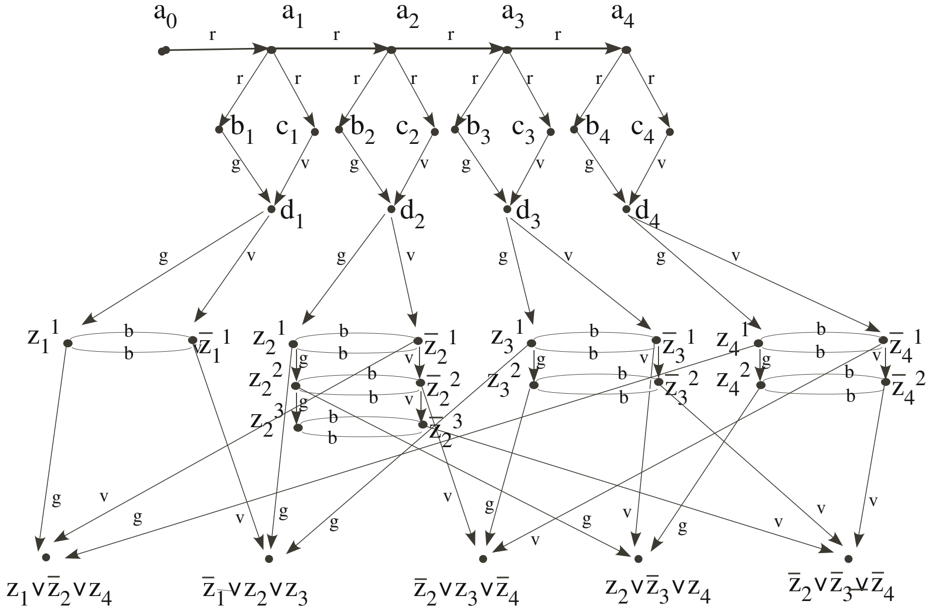


Fig. 1. The graph $G = (N, A)$ corresponding to an instance I of MINIMUM ONES having as collection of clauses the set $C = \{z_1 \vee \overline{z}_2 \vee z_4, \overline{z}_1 \vee z_2 \vee z_3, \overline{z}_2 \vee z_3 \vee \overline{z}_4, z_2 \vee \overline{z}_3 \vee z_4, \overline{z}_2 \vee \overline{z}_3 \vee \overline{z}_4\}$

trivial solution, otherwise, if $d(T) < M$, the algorithm would use T to construct and return a solution Z' for instance I having $|Z'| = d(T)$.

Let us verify that this algorithm approximates MINIMUM ONES within $n^{1-\bar{\varepsilon}}$. If I is not satisfiable this algorithm obviously returns the trivial solution and, following [8] (pag. 254), the behavior of an approximation algorithm is not measured in this case. If I is satisfiable it follows easily that $opt(I') \leq n$; it must also be that $d(T) < M$, otherwise the inequality $d(T) \leq opt(I')\alpha\sqrt[3]{n'^{1-\bar{\varepsilon}}} \leq opt(I')\alpha\sqrt[3]{\beta n^3} \leq opt(I')\, n \leq n^2$ would contradict the inequality $n^2 < M \leq d(T)$. Hence in this case the algorithm uses T to construct and return a solution Z' having $|Z'| = d(T)$ and, as observed at the end of the reduction, in this case we know that $opt(I) = opt(I')$. Hence we can derive the following inequalities

$$\frac{|Z'|}{opt(I)} = \frac{d(T)}{opt(I')} \leq \alpha\sqrt[3]{n'^{1-\bar{\varepsilon}}} \leq \alpha\sqrt[3]{(\beta n^3)^{1-\bar{\varepsilon}}} \leq n^{1-\bar{\varepsilon}}$$

which concludes the proof. □

3 Formulations

The MINCCA problem has the same feasible region of the Minimum Weight Rooted Arborescence problem: The set of spanning arborescences. The latter

problem is solvable in polynomial time [9]. It can be solved in polynomial time also by Linear Programming, since we know its convex hull. For any node i of G, with $\delta^+(i)$ and $\delta^-(i)$ we denote the set of outgoing arcs and the set of incoming arcs, respectively.

Proposition 1. (Schrijver [10], pg. 897) *Let $G = (N, A)$ be a digraph, r a vertex in N such that G contains a spanning arborescence rooted at r. Then the r-arborecence polytope of G is determined by:*

$$\left\{ x \mid x \geq 0, \sum_{e \in \delta^+(X)} x_e \geq 1 \quad \forall X \subset N \text{ with } r \in X, \sum_{e \in \delta^-(i)} x_e = 1 \quad \forall i \in N \setminus \{r\} \right\}.$$

Therefore, to formulate MinCCA as an Integer Program we can use the same polyhedral description of the feasible region. However, since we have a quadratic objective function, the continuous relaxation is no longer integral.

The MinCCA problem has the following Bilinear Integer Program:

$$\min \quad \sum_{i \in N \setminus \{r\}} \sum_{a \in \delta^-(i)} \sum_{b \in \delta^+(i)} d_{c(a),c(b)} x_a x_b \tag{1}$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(X)} x_a \geq 1, \qquad \forall X \subset N, r \in X, \tag{2}$$

$$\sum_{e \in \delta^-(i)} x_e = 1, \qquad \forall i \in N \setminus \{r\}, \tag{3}$$

$$x_a \in \{0, 1\}, \qquad \forall a \in A. \tag{4}$$

The objective function (1) takes into account the changeover cost occurring at each node i except the root node r. If both the incoming arc a (with color $c(a)$) and the outgoing arc b (with color $c(b)$) are part of the arborescence, that is the corresponding variables x_a and x_b are equal to 1, we pay for the cost $d_{c(a),c(b)}$. The constraints (2) are the well–known r-cut inequalities, which force every proper subset of N that contains the root node r to have at least one outgoing arc. They are exponential in number, but they can be separated in polynomial time by solving a r-min-cut problem.

The objective function (1) can be linearized in a standard way by introducing binary variables z_{ab} instead of the bilinear term $x_a x_b$, and the linking constraints:

$$z_{ab} \geq x_a + x_b - 1, \quad z_{ab} \leq x_a, \quad z_{ab} \leq x_b. \tag{5}$$

3.1 Combinatorial Lower and Upper Bounds

A simple lower bound to (1)–(4) can be derived by applying the technique used by Gilmore and Lawler to derive their well–known lower bound on the Quadratic Assignment Problem [11]. For a review of their technique in a more general setting, see [12].

Let p_e be the smallest possible contribution to the objective function (1) of arc $e = (i, j)$, if this arc is part of the optimal solution, that is:

$$p_e = p_{(i,j)} = \min_{f \in \delta^-(i)} d_{c(f),c(e)} \tag{6}$$

Computing p_e takes time $O(n)$ or $O(d_i)$, where d_i is the in-degree of vertex i.

Once we have collected the values of p_e for every $e \in A$, we can solve the Minimum Cost Arborescence problem (with root node $r = 1$), where the arc costs are set to p_e in time $O(m + n \log n)$ [13]. The cost of the optimal solution of this Minimum Cost Arborescence problem provides a lower bound on the optimal value of MinCCA, whereas the changeover cost of such solution yields an upper bound.

3.2 A Stronger Linearization

The continuous relaxation of the basic linearization of (1)–(4) given by variables z_{ab} and by the linking constraints (5) provides rather loose lower bounds. A stronger linearization can be obtained by applying some of the techniques presented in [12] and [14]. We have evaluated one of these techniques, where each constraint of the form $a^T x \leq b$ is multiplied on both sides by a variables x_i. Since for $\{0, 1\}$ variables we have that $x_i^2 = x_i$, the constraint becomes:

$$\sum_{j \neq i} a_j x_j x_i \leq (b - a_i) x_i,$$

and by using the existing z_{ij} variables (and eventually introducing missing variables), we get its linearized version:

$$\sum_{j \neq i} a_j z_{ij} \leq (b - a_i) x_i.$$

If we apply this technique to constraints (2), we do not get stronger relaxations, since the constraints have the form $a^T x \geq b$, and all coefficients are equal to one. Hence, we would get $\sum_{j \neq i} x_j x_i \geq 0$, that is, a redundant constraint.

This technique is more effective when applied to (3) and to the following redundant constraint:

$$\sum_{a \in A} x_a = n - 1. \tag{7}$$

If we apply this technique to (7), we get the quadratic constraints

$$\sum_{a \in A, a \neq b} x_a x_b = (n - 2) x_b, \quad \forall b \in A,$$

and by using the z_{ab} variables, their linearized version:

$$\sum_{a \in A, a \neq b} z_{ab} = (n - 2) x_b, \quad \forall b \in A. \tag{8}$$

If the variables z_{ab} were already introduced to linearize (1) they can be reused. Otherwise new variables z_{ab} need to be introduced, along with their corresponding linking constraints given in (5).

Similarly, by applying this technique to (3), we obtain the linearized version:

$$\sum_{a \in \delta^-(i), a \neq b} z_{ab} = 0, \quad \forall i \in N \setminus \{r\}, \forall b \in \delta^-(i). \tag{9}$$

4 Heuristic Algorithm

Let digraph $G = (N, A)$ be as described in the Introduction and let $A_i \subset A$ be the set of arcs having the same i-th color, out of the set $C = \{1, \dots, k\}$ of colors. For each $i \in C$ consider the digraph $G_i = (N, A_i)$ and let (N_i, T_i) be a largest (non-spanning) arborescence of G_i rooted at node 1, that is a largest set of nodes N_i reachable from node 1 in G_i. Denote the cardinality of N_i by σ_i. We describe now a constructive greedy heuristic for the MinCCA problem.

Algorithm 1. MinCCA Greedy Heuristic

1: **for** i:=1 **to** k **do** compute σ_i;
2: let j be such that $\sigma_j = \max\{\sigma_1, \dots, \sigma_k\}$;
3: $T := T_j$;
4: **while** $(|T| \neq n - 1)$ **do**
5:     delete from A all arcs in $A \setminus T$ entering nodes already reached from 1 in (N, T);
6:     **forall** $h := 1, \dots, k$ **do**
7:         let A_h be the set of arcs in $A \setminus T$ colored h;
8:         find in $(N, T \cup A_h)$ the (not necessarily spanning) arborescence (N, T_h) rooted at node 1, that has minimum changeover cost d_h among those having maximum cardinality;
9:         $c_h := \frac{d_h}{|T_h| - |T|}$;
10:     **end do**
11:     let j be such that $c_j = \min\{c_1, \dots, c_k\}$;
12:     $T := T_j$;
13: **end while**
14: **return** (N, T);

In order to prove that this algorithm is polynomial we only need to show that the very particular case of MinCCA in line 8 is polynomially solvable. In fact, let G' be the subgraph of $(N, T \cup A_h)$ induced by the subset $N' \subseteq N$ of nodes reachable from node 1. Consider any arc a of G' of color h having its tail in a node already reached from 1, with incoming arc b of T, and set the cost of this arc equal to $d_{c(b),h}$. Set the cost of all other arcs of G' to zero. The minimum cost spanning arborescence of G' rooted in node 1 gives T_h.

Improvements to the solution returned by the algorithm can be obtained by applying standard Local Search procedures.

5 Computational Results

The lower bounds presented in Section 3 and the greedy heuristic described in Section 4 are evaluated using a wide collection of instances. These instances are generated using the data in the SteinLib, which is a collection of Steiner tree problems on graphs (`http://steinlib.zib.de/`). In particular, we have used the data sets named `B`, `I080`, `es10fst`, and `es20fst`. Since the library contains undirected graphs, we first introduced for each edge $\{i, j\}$ two arcs (i, j) and (j, i). Each new arc gets a color randomly drawn from $\{1, \ldots, k\}$, where the number k of colors ranges in the set $\{2, 3, 5, 7\}$. We consider two types of costs: *uniform* costs, i.e., all costs equal to 1 whenever there is a change of color, and *non-uniform* costs, randomly drawn from $\{1, \ldots, 10\}$.

We implemented all the algorithms in C++, using the g++ 4.3 compiler and the CPLEX12.2 callable library. Constraints (2) are separated using the MinCut algorithm by Hao and Orlin [15], and implemented as a CPLEX lazy cut callback. Constraints (8) and (9) are defined as lazy constraints and they are handled by CPLEX. The tests are run on a standard desktop with 2Gb of RAM and a i686 CPU at 1.8GHz. All instances used for the experiments are available at `http://www-dimat.unipv.it/~gualandi/Resources.html`.

5.1 Lower Bounds on Small Instances

The first set of results allow us to compare the strength of the three lower bounds presented in Section 3. Let LB_1 be the simple combinatorial lower bound (6), LB_2 be the one obtained solving the continuous relaxation (1)–(5), and LB_3 be the one obtained with the continuos relaxation (1)–(5) plus the linearized quadratic constraints (8) and (9).

Table 1 shows the bounds obtained on 10 small instances. These instances show that there is not a strict dominance between the three lower bounds. However, as we would expect, LB_3 dominates LB_2. Interestingly, in four instances (i.e., instances `ex-0`, `ex-1`, `ex-3`, and `ex-7`), the continuous relaxation used for LB_3 gives an integral solution that hence equals the optimum.

Table 1. Comparison of lower bounds on small instances

Instance	n	m	k	Opt	LB_1	LB_2	LB_3
ex-0	7	11	6	17	13	15.5	**17**
ex-1	7	12	6	2	**2**	1.5	**2**
ex-2	7	20	6	6	0	3	3.3
ex-3	7	10	6	11	9	10	**11**
ex-4	7	14	6	11	3	7.3	7.5
ex-5	7	12	6	15	11	10	13
ex-6	7	12	6	12	7	10	11.7
ex-7	7	13	6	11	10	9.5	**11**
ex-8	7	15	6	3	**3**	2	2.5
ex-9	7	16	6	7	3	3.5	4

Table 2. Instances with uniform costs and 5 colors

Istance	n	m	Opt	UG	Gap	UB_1	Gap	LB_1	Gap	LB_2	T_{LB_2}	LB_3	T_{LB_3}	Gap
es10fst10	18	42	12	14	0.17	14	0.17	8	0.33	3.3	0.3	3.3	1.0	0.72
es10fst11	14	26	11	11	0.00	11	0.00	11	0.00	5.0	0.1	5.0	0.1	0.55
es10fst12	13	24	8	8	0.00	8	0.00	6	0.25	3.0	0.1	3.0	0.1	0.63
es10fst13	18	42	9	11	0.22	9	0.00	6	0.33	1.8	0.3	1.8	0.6	0.80
es10fst14	24	64	11	15	0.36	13	0.18	10	0.09	3.7	1.2	3.7	4.3	0.67
es10fst15	16	36	9	9	0.00	11	0.22	7	0.22	5.0	0.2	5.0	0.4	0.44
es20fst10	49	134	28	33	0.18	48	0.45	19	0.32	3.7	19.8	3.7	96.6	0.87
es20fst11	33	72	28	28	0.00	32	0.14	26	0.07	8.6	2.6	8.6	7.1	0.69
es20fst12	33	72	22	25	0.14	32	0.28	18	0.18	6.3	2.7	6.3	4.9	0.71
es20fst13	35	80	20	22	0.10	34	0.55	16	0.20	6.0	2.9	6.0	9.7	0.70
es20fst14	36	88	18	22	0.22	35	0.59	13	0.28	2.5	4.5	2.5	25.4	0.86
es20fst15	37	86	22	22	0.00	36	0.64	16	0.27	6.8	5.2	6.8	17.0	0.69
Averages:					0.12		0.27		0.21		3.3		13.9	0.69

Table 3. Instance with random costs and k colors

Istance	n	m	k	Opt	UG	Gap	UB_1	Gap	LB_1	Gap	LB_2	T_{LB_2}	T_{LB_3}	Gap
es20fst10	49	134	2	*51*	96	0.88	112	1.20	34	0.33	22.0	35.7	61.4	0.57
			3	*73*	119	0.63	101	0.38	19	0.74	2.0	25.1	40.8	0.97
			5	*110*	159	0.45	124	0.13	64	0.42	16.8	17.6	154.8	0.85
			7	*124*	192	0.55	152	0.23	75	0.40	16.8	21.6	128.7	0.86
es20fst11	33	72	2	85	94	0.11	111	0.31	76	0.11	34.0	2.6	6.6	0.60
			3	42	46	0.10	60	0.43	25	0.40	14.0	2.8	3.7	0.67
			5	122	147	0.20	137	0.12	100	0.18	36.0	2.5	5.4	0.70
			7	96	111	0.16	101	0.05	72	0.25	31.3	2.6	5.1	0.67
es20fst12	33	72	2	78	78	0.00	86	0.10	44	0.44	9.0	3.5	3.5	0.88
			3	55	78	0.42	85	0.55	28	0.49	17.7	2.5	3.4	0.68
			5	95	111	0.17	102	0.07	61	0.36	33.3	3.2	5.3	0.65
			7	86	89	0.03	92	0.07	60	0.30	20.2	2.2	4.6	0.77
es20fst13	35	80	2	62	67	0.08	66	0.06	32	0.48	13.0	5.0	5.3	0.79
			3	102	102	0.00	103	0.01	69	0.32	0.0	3.3	8.7	1.00
			5	92	96	0.04	108	0.17	45	0.51	22.3	3.1	7.6	0.76
			7	107	119	0.11	109	0.02	74	0.31	39.8	4.1	6.7	0.63
es20fst14	36	88	2	40	58	0.45	67	0.68	22	0.45	0.0	4.2	15.2	1.00
			3	54	66	0.22	65	0.20	25	0.54	7.5	6.6	14.7	0.86
			5	78	114	0.46	103	0.32	47	0.40	10.3	5.1	22.6	0.87
			7	103	135	0.31	142	0.38	53	0.49	15.2	5.8	20.9	0.85
es20fst15	37	86	2	55	69	0.25	83	0.51	19	0.65	0.0	5.9	6.5	1.00
			3	56	60	0.07	67	0.20	27	0.52	11.0	5.0	19.3	0.80
			5	110	114	0.04	126	0.15	77	0.30	36.0	6.0	12.2	0.67
			7	112	152	0.36	115	0.03	86	0.23	24.7	5.8	15.4	0.78
Averages:						0.25		0.26		0.40		7.57	24.1	0.79

5.2 Lower Bounds on Medium Size Instances

On bigger instances the strength of the three lower bounds changes, and the combinatorial lower bound dominates the other two: it provides tighter lower bounds, while requiring a negligible computation time.

Tables 2 and 3 show, in addition to the three lower bounds, the upper bound found with the greedy heuristic (UG) and the upper bound given by the arborescence found while computing the combinatorial lower bound (UB_1). For each instance, the tables report the number of nodes, arcs and colors, and the optimum solution "Opt". When we do not know the optimum values, as for instance for the `es20fst10` instances, they appear in italics. For LB_2 and LB_3 we report the computational time in seconds, called T_{LB_2} and T_{LB_3}, respectively. In the following, for each bound b we define its *gap* from the optimum as the ratio $\frac{|Opt-b|}{Opt}$.

Table 2 shows the results on instances with uniform costs and 5 colors. The combinatorial lower bound LB_1 is always better than the LP relaxation bounds LB_2 and LB_3; the average gap for LB_1 is 0.21%, while the average gap for LB_2 and LB_3 is 0.69%. Differently from the case of small instances, here LB_2 and LB_3 have always the same value, contradicting what we would expect from previous work [12, 14]; however the computational time to obtain LB_2 is much lower than the one to obtain LB_3, that is, T_{LB_2} is, on average, 3.3 sec. while T_{LB_3} is 13.9 sec.. Notice finally that, for this class of instances, the greedy heuristic finds upper bounds that have an average gap of 12%.

Table 3 shows the results on instances derived from the `es20fst` data set, using a various number of colors and costs randomly drawn in the set $\{1, \ldots, 10\}$. These

Table 4. Results of the branch-and-cut algorithm: Easy instances

Istance	n	m	Opt	Cuts	Nodes	Time	Cuts	Nodes	Time
es20fst11	33	72	122	188	723	12	147	667	35
			42	115	277	7	80	348	15
			85	374	1252	19	435	1048	55
			96	103	741	10	113	790	37
es20fst12	33	72	55	86	191	6	112	227	15
			78	63	327	7	62	354	34
			95	105	414	8	122	415	27
			86	57	772	9	55	708	39
es20fst13	35	80	102	58	280	8	60	732	63
			92	42	244	7	48	392	39
			62	112	843	16	118	920	64
			107	42	665	12	52	597	74
es20fst14	36	88	78	94	1395	35	104	1464	252
			40	86	850	35	88	692	186
			54	98	1376	41	92	737	119
es20fst15	37	86	110	242	2233	45	330	2073	174
			55	549	3344	71	592	5246	509
Averages:				142	937	20	154	1024	102

results show, unfortunately, that on these instances both the lower and upper bounds are weak. The upper bounds obtained while computing the combinatorial lower bound are sometimes better than those obtained with the greedy heuristic.

5.3 Branch-and-Cut

Finally, we present the results obtained with an exact branch-and-cut algorithm that embeds the combinatorial lower bound and the greedy heuristic; we compare these results using the two linearizations proposed in Section 3.

Table 5. Results of the branch-and-cut algorithm: Hard instances

Istance	n	m	Opt	Cuts	Nodes	Time	Cuts	Nodes	Time
es20fst10	49	134	*152*	239	5168	-	23	183	-
			116	223	4465	-	23	183	-
			96	96	1650	-	113	267	-
			101	321	3572	-	55	163	-
s20fst14	36	88	103	155	4264	93	155	3186	-
es20fst15	37	86	56	586	4177	99	556	5312	-
			112	629	5978	118	645	5931	-

Table 6. Challenging instances with uniform costs

Instance	n	m	Best	UG	UB_1	LB_1	Gap	LB_2	Gap	T_{LB_2}
b01	50	126	28	35	34	18	0.36	11.7	0.58	21
b02	50	126	30	31	34	20	0.33	8.8	0.71	15
b03	50	126	28	35	36	15	0.46	5.3	0.81	19
b04	50	200	24	26	35	6	0.75	1.6	0.93	66
b05	50	200	18	21	31	5	0.72	3.1	0.83	68
b06	50	200	20	23	31	4	0.80	1.0	0.95	65
b07	75	188	53	53	54	31	0.42	13.4	0.75	83
b08	75	188	51	55	56	31	0.39	13.8	0.73	59
b09	75	188	47	50	47	29	0.38	13.5	0.71	75
b10	75	300	36	36	46	8	0.78	3.5	0.90	212
b11	75	300	39	39	52	8	0.79	1.3	0.97	177
b12	75	300	35	35	44	5	0.86	1.2	0.97	179
b13	100	250	62	65	62	31	0.50	22.1	0.64	168
b14	100	250	59	59	70	33	0.44	18.7	0.68	150
b15	100	250	68	68	71	36	0.47	19.2	0.72	154
b16	100	400	42	42	63	6	0.86	0.3	0.99	1143
b17	100	400	46	46	63	7	0.85	1.2	0.97	350
b18	100	400	54	54	66	7	0.87	2.3	0.96	436
I080-001	80	240	47	47	59	28	0.40	15.2	0.68	159
I080-002	80	240	49	49	58	20	0.59	8.0	0.84	173
I080-003	80	240	43	43	46	24	0.44	7.3	0.83	131
I080-004	80	240	48	48	51	22	0.54	10.1	0.79	139
I080-005	80	240	41	41	56	13	0.68	6.0	0.85	228
Averages:							0.60		0.82	

Tables 4 and 5 report the results for the es20fst data set. Table 4 considers only those instances that both linearizations where able to solve within a time limit of 1000 seconds (in order to avoid censored data in the averages). Table 5 shows the results for the instances in which at least one of the two versions reached the time limit. As we would expect after the results shown in Tables 2 and 3, the branch-and-cut algorithm based on the standard linearization is much faster than the linearization obtained through a quadratic reformulation.

To conclude, Table 6 presents the results obtained on two other data sets, namely, the B data set and the I080 data set. Though these data sets are easy for Steiner tree problems, they are demanding for the MinCCA problem. The column "Best" reports the best upper bounds we were able to compute with our branch-and-cut algorithm, with a timeout of 1000 seconds. We propose these instances as challenging instances of the MinCCA problem.

References

1. Wirth, H., Steffan, J.: Reload cost problems: minimum diameter spaning tree. Discrete Applied Mathematics 113, 73–85 (2001)
2. Galbiati, G.: The complexity of a minimum reload cost diameter problem. Discrete Applied Mathematics 156, 3494–3497 (2008)
3. Gamvros, I., Gouveia, L., Raghavan, S.: Reload cost trees and network design. In: Proc. International Network Optimization Conference, paper n.117 (2007)
4. Amaldi, E., Galbiati, G., Maffioli, F.: On minimum reload cost. Networks (to appear)
5. Gourvès, L., Lyra, A., Martinhon, C., Monnot, J.: The minimum reload s-t path, trail and walk problems. In: Nielsen, M., Kučera, A., Miltersen, P.B., Palamidessi, C., Tůma, P., Valencia, F. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 621–632. Springer, Heidelberg (2009)
6. Kann, V.: Polynomially bounded minimization problems that are hard to approximate. Nordic J. Comp. 1, 317–331 (1994)
7. Jonsson, P.: Near-optimal nonapproximability results for some NPO PB-complete problems. Inform. Process. Lett. 68, 249–253 (1997)
8. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M.: Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties. Springer, Heidelberg (1999)
9. Edmonds, J.: Optimum branchings. J. Res. Nat. Bur. Stand. (B) 71, 233–240 (1967)
10. Schrijver, A.: Combinatorial optimization: polyhedra and efficiency. Springer, Heidelberg (2003)
11. Lawler, E.: The quadratic assignment problem. Manag. Sci. 9(4), 586–599 (1963)
12. Caprara, A.: Constrained 0–1 quadratic programming: Basic approaches and extensions. European Journal of Operational Research 187, 1494–1503 (2008)
13. Gabow, H., Galil, Z., Spencer, T., Tarjan, R.: Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica 6(2), 109–122 (1986)
14. Buchheim, C., Liers, F., Oswald, M.: Speeding up ip-based algorithms for constrained quadratic 0–1 optimization. Math. Progr. (B) 124(1-2), 513–535 (2010)
15. Hao, J., Orlin, J.: A faster algorithm for finding the minimum cut in a graph. In: Proc. of the 3rd ACM-SIAM Symposium on Discrete Algorithms, pp. 165–174 (1992)

Localizing Program Logical Errors Using Extraction of Knowledge from Invariants

Mojtaba Daryabari, Behrouz Minaei-Bidgoli, and Hamid Parvin

School of Computer Engineering, Iran University of Science and Technology (IUST),
Tehran, Iran
{daryabari,b_minaei,parvin}@iust.ac.ir

Abstract. Program logical error localization and program testing are two of the most important sections in software engineering. Programmers or companies that produce programs will lose their credit and profit effectively if one of their programs delivered to a customer has any drawback. Nowadays there are many methods to test a program. This paper suggests a framework to localize the program logical errors by extraction of knowledge from invariants using a clustering technique.

Keywords: Daikon, Invariant, Dynamic Invariant Detection, Variable Relations, Program point, Software engineering, Verification.

1 Introduction

Program error localization is one of the most important fields in program production and software engineering. Programmers or companies that produce programs will lose their credit and profit effectively if one of their programs delivered to a customer has any drawback. While there are many methods to test a program, it is the lack of an appropriate method for localizing a program logical errors using extraction of knowledge from invariants (Catal, 2011). This paper will offer a framework to localize the program logical errors before its release. This method is based on extracting the knowledge from program invariants called Logical Error localizator Based on Program Invariants (LELBPI).

Invariants are a set of rules that govern among the values of variables in the programs in such a way that they remain unchanged in the light of different values of the input variables in the consecutive runnings of a program. There are three types of invariant generally: pre-condition, loop-invariant and post-condition. However post-conditions are considered as a kind of invariants. In this paper, where it is addressed invariant, it only implies to the invariant of post-condition type.

This paper suggests a framework to localize program logical errors with the use of a series of tools such as Daikon which derives program invariants (Ernst et al. 2000). There are many tools to extract program invariants that use static or dynamic methods. The some invariant-extractor methods will be explained in following section.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 124–135, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Briefly, the used method is to first collect a repository of the evaluated programs. Then using their reliabilities of the invariants assigned by an expert the programs are clustered. The clustering is done regarding to their invariants likeness. For example, all types of a sorting program including bubble sort, merge sort, insertion sort and etc and their invariants stand in the same cluster. LELBPI checks its tested program invariants with all sets of cluster invariants that are available in its repository. After that LELBPI calculates their similarity measures with the clusters and selects cluster with maximum similarity. If difference number with one set of the clusters invariants in the repository is zero then the program will be true else it will be a new one or belongs to another of the pre-defined clusters; besides it has some error(s) that must be eliminated.

2 Invariants

Invariants in programs are formulas or rules that are emerged from source code of program and remain unique and unchanged with respect to running of program with different input parameters. For instance, in a sort program that its job is to sort array of integers, the first item in the array must be bigger than the second item and the second item must be bigger than the third, etc. Invariants have significant impact on software testing. Daikon is the suitable software for dynamic invariant detection developed until now in comparing other dynamic invariant detection methods. however this method has some problems and weaknesses and thus, many studies have been carried out with the aim of improving Daikon performance which have resulted in several different versions of Daikon up to now (Ernst et al. 2000), (Perkins and Ernst 2004). For instance latest version of Daikon includes some new techniques for equal variables, dynamically constant variables, variable hierarchy and suppression of weaker invariants (Perkins and Ernst 2004).

Invariants in programs are sets of rules that govern among the values of variables and remain unchanged in the light of different values of the input variables in consecutive runnings of the program. Invariants are very useful in testing software behavior, based on which a programmer can conclude that if its program behavior is true (Kataoka et al. 2001), (Nimmer and Ernst 2002). For instance, if a programmer, considering invariants, realizes that the value of a variable is unwillingly always constant, s/he may conclude that its codes have some bugs.

Also, invariants are useful in comparing two programs by programmers and can help them check their validity. For instance, when a person writes a program for sorting a series of data, s/he can conclude that his program are correct or has some bugs by comparing his program invariants against the invariants of a famous reliable sort program; such as Merge Sort. Here, the presupposition is that in two sets (a) invariants detected in the program and (b) invariants detected in the merge sort program, must be almost the same. Additionally, invariants are useful in documentation and introduction of a program attributes; i.e. in cases where there are no documents and explanations on a specific program and a person wants to recognize its attributes for correcting or expanding program, invariants will be very helpful to attain this goal, especially if the program is big and has huge and unstructured code.

There are two ways for invariant detection that are called *static* and *dynamic*. In the static way, invariants are detected with the use of techniques based on compiler issues (for example, extraction of data flow graphs of the program source code). Dynamic way, on the other hand, detects invariants with the help of several program runnings by different input parameter values and based on the values of variables and relations between them. Dynamic methods will be explained in more detail in next section (Ernst et al. 2006).

Every method has some advantages and disadvantages which will be debated in this paper. There are some tools as Key & ESC for java language and LClint for C language for static invariant detection (Evans et al. 1994), (Schmitt and Benjamin 2007). In static detection, the biggest problem is the difficulty with which a programmer can discover the invariants. Tracing of codes and detection of rules between variable values are a difficult job especially if the programmer wants to consider such cases as pointers, polymorphisms and so on.

In dynamic methods, the biggest problem is that they are careless and time-consuming and, more importantly, do not provide very reliable answers.

3 Related Works

3.1 Background

There are many machine learning based and statistical based approaches to fault prediction. Software fault prediction models have been studies since 1990s until now (Catal, 2011). According to recent studies, the probability of detection (71%) of fault prediction models may be higher than probability of detection of software reviewer (60%) if a robust model is built (Menzies et al., 2007).

Software fault prediction approaches are much more cost-effective to detect software faults compared to software reviews. Although benefits of software fault prediction are listed as follows (Catal and Diri, 2009):

- Reaching a highly dependable system
- Improving test process by focusing on fault-prone modules
- Selection of best design from design alternative using object-oriented metrics
- Identifying refactoring candidates that are prediction as fault-prone
- Improving quality by improving test process

3.2 Daikon Algorithm

Daikon first runs program with several different input parameters. Then it instruments program and finally in every running of the program saves variable values on a file called data trace file. Daikon continues its work with extracting the values of variables from data trace files and by use of a series of predefined relations discovers the invariants and saves them. Daikon discovers unary, binary and ternary invariants. Unary invariants are invariants defined on one variable; for instance, X>a presents variable X is bigger than a constant value. For another example X (mod b)=a shows

X mod b=a. X>Y, X=Y+c are also samples of binary invariants and Y=aX+bZ is a sample of ternary invariant considered in Daikon in which X, Y, Z are variables and a & b are constant values.

Daikon will check invariants on the next run of the program on the data trace file and will throw them out from list of true invariants if it is not true on current values of variables. Daikon continues this procedure several times while concluding proper reliability of invariants (Ernst et al. 2006).

4 Suggested Framework

Since the similar programs with the same functionalities have more or less the same invariants, so invariants can be considered as behaviors of the programs. Although it is highly probable that there is a program with job similar to job of another program plus an auxiliary job, these two programs are not considered as the same programs. This is due to high rate differences in their invariants. In other words, if the two programs just do the same job, their invariants are almost the same. Suggested framework is explained in this section. Informally, the LELBPI is shown in Fig 1.

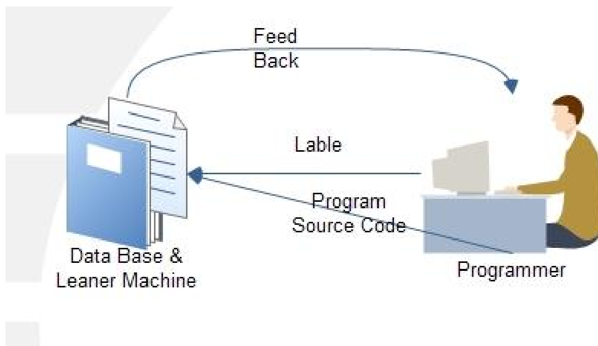


Fig. 1. Suggested framework for LELBPI

Fig 1 depicts that program invariants detected by Daikon software are sent to LELBPI and machine processes them and give a feedback to programmer. Then a label indicating the validation of LELBPI prediction is returned to it. It is notable that the rules included in the Data Base of the learner machine must be in the general forms. For example to show that an array is sorted, the corresponding rule is similar to "*array_name sorted by* >=" (denoted by *rule 1*); besides the invariants similar to "*array_name[1]<= array_name[2]*" (denoted by *rule 2*) is eliminated provided that the *rule 1* is available. Details of framework are illustrated in activity diagram which is shown in Fig 2.

First, program invariants are sent to LELBPI then invariants are compared with all of clusters agent and select cluster with minimum difference. If all of cluster agents are far away then it will be asked from programmer that "This program is unknown, is

it new and reliable?" If user answers that it is ok and s/he is sure that program is true, nevertheless invariants will insert to database which locates in LELBPI and updates its clusters.

While entered invariants by user is resemble with some cluster, matter of cluster which has minimum difference will be shown to user and if user confirms it, provided that difference number is zero (or be less than a pre-defined threshold) then tested program invariants insert to database which locates in machine learner and updates machine learner clusters, else if difference number is more than zero (or be greater than the pre-defined threshold) then just sends an alarm to user which program has some logical error(s).

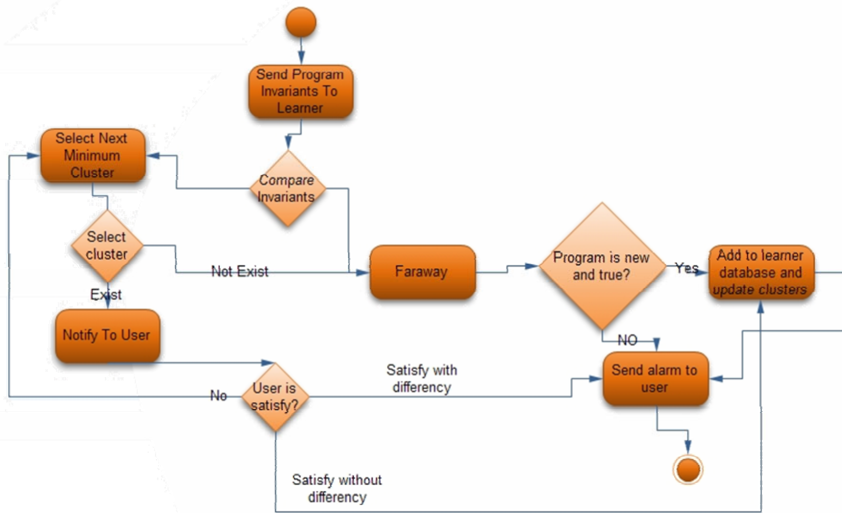


Fig. 2. Error localizator framework activity diagram

While presented cluster to user dissatisfies programmer then LELBPI selects the next minimum cluster and sends its matter to user. If all similar clusters presented to user can not satisfy programmer then asks of user again that "This program is unknown, is this program new and reliable?" Again if user answers that it is ok and s/he is sure that program is true, nevertheless invariants will insert to database located in LELBPI and updates its clusters.

It is clear that learner machine database is empty initially and it will gradually be filled by adding true and reliable programs manually by user feedbacks. The greater the number of record in database, the more accurate and valid results of suggested framework.

4.1 Variable Matching

Suppose that the agent of cluster that must be compared with the tested program invariants had two integer variables denoted by *i* and *j*. Also assume that entered

invariable to learner machine had two different integer variable denoted by *m*, *n*. Before than compare invariants, LELBPI must match either $i \rightarrow m$ and $j \rightarrow n$ or vice versa. Algorithm that is used in LELBPI matches all possible permutations of *i, j* to *m, n* and amount of difference on each combination will be computed and software will select combination which has minimum difference in compare with other combination. More detailed explanation is offered in section five.

For decreasing runtime, LELBPI standardizes variable names before than invariants of true program are added to database, it means that all variables with the same data type rename to one series of standard and recorded name. For example all integer variables in all programs are renamed to *int_a*, *int_b*... sequentially and all integer arrays in programs are renamed to *array_int_a*, *array_int_b*… sequentially, and for each invariant set, the numbers of variables in all data types are buffered. Machine can permute variables with higher speed.

5 Scalability

For justifying this method is commodious, run time of this method is estimated on this section. Informally runt time can be characterized as:

$$\text{Time=O}\left(Inv + K \times \binom{Max(f,r)}{Min(f,r)} \times m \times g \right),$$

where *Inv* is run time of extracting tested program invariants, *K* is cluster number, *f* is maximum of number of variables in one type in tested program, *r* is maximum of number of variables in one type in agent of clusters, *m* is maximum of invariants number in agent of clusters, and finally *g* is maximum of invariants number in tested program invariants.

Pay attention that *f* and *r* is limited because as is said in section 3.1, before than adding invariants in database variables are standardized. So, run time is commodious and acceptable.

6 The Experimental Result

For validating this framework, software is implemented and their results are shown in this section. Invariants of six true programs including bubble sort, merge sort, insertion sort, shell sort, compute sum of array elements and search into array are added to software database initially. Predominate algorithms are shown in below.

The results of pre-mentioned algorithms with this supposition that every used array in programs have random values collected in Table 1. It is necessary to note that all of invariants in this paper are gotten from Daikon software.

In Table 1 "a sorted by <=" means that *a[i]<=a[i+1]*. Also "a=orig (a [])" means that "a" array elements remain unmodified in exit of program.

With respect to invariants in Table 1, two clusters are created and programs are grouped in those which these clusters are sort and search. Agent of sort cluster is "a sorted by <=" and agent of search cluster is "a=orig (a [])".

```
Void  bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;
  for (i=0;i <= (array_size - 1); i++)
    {
      for (j = 1; j<= (array_size - 1);  j++)
        {
          if (numbers[j-1] > numbers[j])
            {
              temp = numbers[j-1];
              numbers[j-1] = numbers[j];
              numbers[j] = temp;
            }
        }
    }
}
```

```
void insertion_sort(int a[], int length)
{
  int i;
  for (i=0; i < length; i++)
    {
      //Insert a[i] into the sorted sublist
      int j, v = a[i];
      for (j = i - 1; j >= 0; j--)
        {
          if (a[j] <= v)
            break;
          a[j + 1] = a[j];
        }
      a[j + 1] = v;
    }
}
```

```
Void shell_sort(int a[], int length)
 {
   int ciura_intervals[] = {701, 301, 132, 57, 23, 10, 4, 1};
   double extend_ciura_multiplier = 2.3;
   int interval_idx = 0;
   int interval = ciura_intervals[0];
   if (length > interval)
     {
       while(length > interval)
         {
          interval_idx--;
          interval = (int)(interval*extend_ciura_multiplier);
         }
     }
   else
     {
       while(length < interval)
         {
           interval_idx++;
           interval = ciura_intervals[interval_idx];
         }
     }
  while (interval > 1)
    {
     interval_idx++;
     if (interval_idx >= 0)
       {
         interval = ciura_intervals[interval_idx];
       }
     else
       {
        interval = (int)(interval/extend_ciura_multiplier);
       }
     shell_sort_pass(a, length, interval);
    }
}
```

```
int sum(int a[])
{

   float sum = 0;
   for (int i=0; i<size; i++)
      {
        sum = sum + a[i];
      }
   return sum;
}
```

```
int search(int a[], int item)
{
   Int index;
   for (int i=0; i<size; i++)
     {
        if a[i]=item
          {
             j=i
          }
     }
   return j;
}
```

```
void merge(int m, int n, int A[], int B[], int
C[])
{
     int i, j, k;
     i = 0;
     j = 0;
     k = 0;
     while (i < m && j < n)
       {
         if (A[i] <= B[j])
             {
               C[k] = A[i];
               i++;
             }
         else
             {
               C[k] = B[j];
               j++;
             }
         k++;
       }
     if (i < m)
       {
         for (int p = i; p < m; p++)
           {
             C[k] = A[p];
             k++;
           }
       }
     else
       {
         for (int p = j; p < n; p++)
           {
             C[k] = B[p];
             k++;
           }
       }
}
```

```
void shell_sort_pass(int a[], int length,
int interval)
{
   int i;
   for (i=0; i < length; i++)
    {
     int j, v = a[i];
     for (j=i-interval; j >= 0; j-= nterval)
       {
        if (a[j] <= v)
            break;
        a[j + interval] = a[j];
       }
     a[j + interval] = v;
    }
}
```

Now consider below code which is bubble sort that programmer don't check last element of array. As you know, it is a common error in programming. This code has an array that its name is *b* and two integer value with names *m, n*.

Table 1. Programs and their invariants

Row	Program Name	Invariants	
1	Bubble Sort	1. 2. 3.	a sorted by >= i= Length(a) J=Length(a)-1
2	Insertion sort	1. 2.	a sorted by >= i= Length(a)-1
3	Shell Sort	1. 2.	a sorted by >= interval=1
4	Merge Sort	1. 2. 3. 4. 5. 6. 7. 8.	a sored by >= b sorted by >= c sorted by >= i < k j <k k= Length(c) a=orig(a[]) b=orig(b[])
5	Sum of array elements	1. 2.	a =orig(a[]) i= Length(a)-1
6	Search array	1. 2.	a=orig(a[]) i<= Length(a)-1

Invariants for Algorithm 1 that are calculated by Daikon software with this supposition that length of array is six is shown at below:

1. m= Length(b)-1
2. n= Length(b)-1
3. b[0]<=b[1]
4. b[1]<=b[2]
5. b[2]<=b[3]
6. b[3]<=b[4]

Number of differences in invariants of every cluster agent and these invariants are shown in Table 2.

Here, machine just compares invariants which exist on variables in cluster agent. It is because tested program may do anything as well as agent function. So if tested program have any invariants on those variables that are absent in set of variables in comparing agent of cluster then these invariants can't be considered as differences. Also because invariants of an agent include intersection of invariants of all programs

in the cluster, some invariants may be in invariants of programs while they are not in the invariant of its cluster agent. However invariants on variables in cluster agent must be strictly in tested program invariants.

```
void  bubbleSort(int b[], int array_size)
{
  int m, n, temp;
  for (m=0;m < (array_size - 1);  m++)
     {
       for (n = 1; n <= (array_size-1); n++)
        {
          if (b[n-1] >= b[n])
            {
              temp = b[n-1];
              b[n-1] = b[n];
              b[n] = temp;
            }
        }
     }
}
```

Algorithm 1. Faulty varsion of bubble sort

Table 2 presents that tested program stands in sort cluster and it have sort matter. Now tested program will be compared with all of program invariants in sort cluster. Results of these comparisons are collected in Table 3.

Table 2. Compare Invariants

Cluster Name	Variable Matching	Differences
Sort	b[]=a[]	3
Search	b[]=a[]	10

It is clear from Table 3 that tested program has minimum difference from bubble sort in condition that *m* variable be assigned to *i* variable and *n* be assigned to *j*. It is clear that it is a valid result. Terminal result is: program matter is sorting and has some logical errors because number of differences is not zero.

Table 3. Compare tested program invariants with program invariants in sort cluster

Row	Program Name	Variables Matching	Differences
1	Bubble Sort	b[]=a[] m=i n=j	1
2	Bubble Sort	b[]=a[] m=j n=i	3
3	Insertion Sort	b[]=a[] m=i	2
4	Insertion Sort	b[]=a[] n=i	3
5	Merge Sort	b[]=a[] or b[]=b[] m=k n=i	10
6	Merge Sort	b[]=a[] or b[]=b[] m=k n=j	10
7	Merge Sort	b[]=a[] or b[]=b[] m=i n=j	10
8	Merge Sort	b[]=a[] or b[]=b[] m=j n=i	10
9	Merge Sort	b[]=c[] m=i n=k	4
10	Merge Sort	b[]=c[] m=j n=k	4
11	Merge Sort	b[]=c[] m=k n=i	4
12	Merge Sort	b[]=c[] m=k n=j	4
13	Merge Sort	b[]=c[] m=i n=j	4
14	Merge Sort	b[]=c[] m=j n=i	4

7 Conclusion and Further Works

Daikon is a method to discover likely invariants by dynamic methods. Also Daikon's team has been doing many researches about invariants application. They also do some researches to test software base on run program with several different input parameters and extract and check invariants on every run of program. In this paper a new framework based on Daikon, is proposed to incrementally detect errors of different programs. In this framework, one cluster is produced per invariants of each program type. This framework is gradually reinforced.

For future direction of research one can do some filtering on programs in every cluster can reduce comparing effectively. For example as one of filtering, machine can just compare programs which have same variable data type for example have three integer values and two array data types. Another action that can be done is that machine as well as true program invariants can learn from false program invariants and machine learns that programmer where and how have fault in program commonly.

References

1. Catal, C., Diri, B.: Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. Information Sciences 179(8), 1040–1058 (2009)
2. Catal, C.: Software fault prediction: A literature review and current trends. Expert Systems with Applications 38, 4626–4636 (2011)
3. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. ACM Trans. Software Eng. and Methodology (1998a)
4. Cook, J.E., Wolf, A.L.: Event-Based Detection of Concurrency. In: Proc. ACM SIGSOFT Symp. (1998b)
5. Dwyer, M.B., Clarke, L.A.: Data Flow Analysis for Verifying Properties of Concurrent Programs. In: Proc. Second ACM SIGSOFT Symp. (1994)
6. Ernst, M.D., Griswold, W.G., Kataoka, Y., Notkin, D.: Dynamically Discovering Program Invariants Involving Collections. Technical Report UW-CSE-99-11-02 (2000)
7. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming (2006)
8. Evans, D., Guttag, J., Horing, J., Tan, Y.M.: LCLint:A Tool for Using Specification to Check Code. In: Proc. Second ACM SIGSOFT Symp. (1994)
9. Kataoka, Y., Ernst, M.D., Griswold, W.G., Notkin, D.: Automated Support for Program Refactoring Using Invariants. In: Proc. Int'l Conf. Software Maintenance (2001)
10. Lencevicius, R., Ho Èlzle, U., Singh, A.K.: Query-Based Debugging of Object-Oriented Programs. In: Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (1997)
11. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering 33(1), 2–13 (2007)
12. Mitchell, M.: An Introduction to Genetic Algorithms. The MIT Press, Cambridge (1998)
13. Moraglio, A., Kim, Y.H., Yoon, Y., Moon, B.R., Poli, R.: Generalized cycle crossover for graph partitioning. In: GECCO (2006)
14. Nimmer, J.W., Ernst, M.D.: Automatic Generation of Program Specifications. In: Proc. Int'l Symp. Software Testing and Analysis (2002)
15. Perkins, J.H., Ernst, M.D.: Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. In: Proc. ACM SIGSOFT Symp. (2004)
16. Schmitt, H., Weiß, B.: Inferring Invariants by Static Analysis in KeY (2007)

Compressed String Dictionaries

Nieves R. Brisaboa[1,⋆], Rodrigo Cánovas[2,⋆⋆], Francisco Claude[3,⋆⋆⋆], Miguel A. Martínez-Prieto[2,4,⋆⋆,†], and Gonzalo Navarro[2,⋆⋆]

[1] Database Lab, Universidade da Coruña, Spain
[2] Department of Computer Science, University of Chile, Chile
[3] School of Computer Science, University of Waterloo, Canada
[4] Department of Computer Science, Universidad de Valladolid, Spain

Abstract. The problem of storing a set of strings – a *string dictionary* – in compact form appears naturally in many cases. While classically it has represented a small part of the whole data to be processed (e.g., for Natural Language processing or for indexing text collections), recent applications in Web engines, RDF graphs, Bioinformatics, and many others, handle very large string dictionaries, whose size is a significant fraction of the whole data. Thus efficient approaches to compress them are necessary. In this paper we empirically compare time and space performance of some existing alternatives, as well as new ones we propose. We show that space reductions of up to 20% of the original size of the strings is possible while supporting dictionary searches within a few microseconds, and up to 10% within a few tens or hundreds of microseconds.

1 Introduction

String dictionaries arise naturally in a large number of applications. We associate them classically to Natural Language (NL) processing: finding the *lexicon* of a text corpus is the first step in analyzing it [25]. They also arise, together with *inverted indexes*, when indexing text collections formed by NL [2,33].

In those NL applications, there has not been much concern about the size of the dictionary. This is because, in classical NL collections, the dictionary grows sublinearly with the text size: Heaps' law [19] establishes that in a text of length n, the dictionary size is $O(n^\beta)$, for some $0 < \beta < 1$ depending on the type of text. This β value is usually in the range 0.4–0.6 [2], and thus the dictionary of terabyte-size collections should occupy just a few megabytes and would easily fit in the main memory of a commodity PC.

Heaps' law, however, does not model well the reality of Web search engines. Web collections are much less "clean" than text collections whose content quality is carefully controlled. Dictionaries of Web crawls easily exceed

⋆ Funded by Ministry of Science and Innovation of Spain (PGE and FEDER) TIN2009-14560-C03-02 and Xunta de Galicia ref. 09TIC060E.

⋆⋆ Funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

⋆⋆⋆ Funded by the David R. Cheriton scholarships program.

† Funded by Ministry of Science and Innovation of Spain, TIN2009-14009-C02-02.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 136–147, 2011.
© Springer-Verlag Berlin Heidelberg 2011

the gigabytes, due to typos and unique identifiers that are taken as "words", but also for "regular words" from multiple languages. The *ClueWeb09* dataset (`http://boston.lti.cs.cmu.edu/Data/clueweb09`; thanks to Leonid Boystov) is a real example which comprises close to 200 million different words obtained from 1 billion web pages on 10 languages. This results in a large dictionary of far more than 1GB.

Web graphs are another application where the size of the URL names, classically neglected, is becoming very relevant with the advances of the techniques that compress the graph topology. The nodes of a Web graph are typically the pages of a crawl, and the edges are the hyperlinks. Typically there are 15 to 30 links per page. Compressing Web graphs has been an area of intense study, as it permits caching larger graphs in main memory, for tasks like Web mining, Web spam detection, finding communities of interest, etc. [21,9]. URL names are used to improve the mining quality [34,27].

In an uncompressed graph, 15 to 30 links per page would require 60 to 120 bytes if represented as a 4-byte integer. This posed a more serious memory problem compared to the name of the URL itself once some simple compression procedure was applied to those names (such as Front-Coding, see Section 2). For example, Broder et al. [5] reports 27.2 bits per edge (bpe) and 80 bits per node (bpn). This means that each node takes around 400–800 bits to represent its links, compared to just 80 bits used for storing its URL. In the same way, an *Internet Archive* graph of 115M nodes and 1.47 billion edges required [31] 13.92 bpe plus around 50 bpn, so 200–400 bits are used to encode the links and only 50 for the URL. In both cases, the space required to encode the URLs was just 10%-25% of that required to encode the links. However, the advances in compressing the edges have been impressive in recent years, achieving compression ratios around 1–2 bits per edge [3,1]. At this rate, the edges leaving a node require on average 2 to 8 bytes, compared to which the name of the URL certainly becomes an important part of the overall space.

Another application is Bioinformatics. Popular alignment software like BLAST [18] indexes all the different substrings of length q of a text, storing the positions where they occur in the sequence database. For DNA sequences $q = 11, 12$ is common, whereas for proteins they use $q = 3, 4$. Over a DNA alphabet of size 4, or a protein alphabet of size 20, this amounts to up to 200 million characters. Using a larger q would certainly allow one improve the quality in searching for conserved regions, but this is infeasible for memory constraints.

The emergent *Linked Data Project* (`http://linkeddata.org`) focuses on the publication of RDF (`http://www.w3.org/TR/rdf-syntax-grammar`) data and their connection between different data sources in the "Web of Data". This movement results in huge and heterogeneous RDF datasets from diverse fields.

The dictionary is an essential component in the logical division of an RDF database [10]. However, its effective representation has not been studied in depth. Our experience with the tool *HDT-It!* (`http://code.google.com/p/hdt-it`) shows that the dictionary for dataset *DBpedia-en* (`http://downloads.dbpedia.org/3.5.1/en`) takes about 80% of the total size.

Finally, Internet routing poses another interesting problem on dictionary strings. Domain name servers map domain names to IP addresses, and routers map IP addresses to physical addresses. They may handle large dictionaries of domain names or IP addresses, and serve many requests per second.

This short tour over various example applications shows that handling very large string dictionaries is an important and pervasive problem. Curiously, we have not seen much research on compressing them, perhaps because a few years ago the space of these dictionaries was not a serious problem, and at most Front-Coding was sufficient. In this paper we study Front-Coding and other solutions we propose for compressing large string dictionaries, so that two basic operations are supported: (1) given a string, return its position in the dictionary or tell it is not in the dictionary; (2) given a position, retrieve its string content.

Our study over various application scenarios spots a number of known and novel alternatives that dominate different niches of the space/time tradeoff map. The least space-consuming variants perform efficiently while compressing the dictionary to 9%–22% of its original size, depending on the type of dictionary.

2 Basic Concepts and Related Work

Rank and select on bitmaps. Let $B[1,n]$ be a $0,1$ string (*bitmap*) of length n and assume there are m ones in the sequence. We define $rank_b(B,i)$ as the number of occurrences of bit b in $B[1,i]$ and $select_b(B,i)$ as the position of the i-th occurrence of b in B.

In this paper we will use two different succinct data structures (implementations available at `http://libcds.recoded.cl`) that answer *rank* and *select* queries. The first one, that we will refer to as *RG* [16], uses $(1+x)n$ bits to represent B. It supports *rank* using two random accesses to memory plus $4/x$ contiguous (i.e., cached) accesses. *Select* requires and additional binary search.

The second data structure, that we will call *RRR* [29], is a compressed bitmap that uses in practice about $\log\binom{n}{m}+(\frac{4}{15}+x)n$ bits (our logarithms are in base 2), answering *rank* within two random accesses plus $3+8/x$ accesses to contiguous memory, and *select* with an extra binary search. In practice this compresses the bitmap when $m<0.2\,n$.

Huffman and Hu-Tucker codes. For compressing sequences, statistical methods assign shorter codes (i.e., bit streams) to more frequent symbols. Huffman coding [20] is the optimal code (i.e., it achieves the minimum length of encoded data) that is uniquely decodable. In this paper we use canonical Huffman codes [26], which have various advantages.

Hu-Tucker codes [22] are optimum among those that maintain the lexicographical order of the symbols. Two sequences encoded using Hu-Tucker can be lexicographically compared bytewise directly in encoded form. We use both codes in this paper, in some cases padding them (with zeros) to the next byte in order to simplify alignment and bytewise comparisons.

Hashing. Hashing [8] is a folklore method to store a dictionary of any kind. A *hash function* transforms the elements into indexes in a *hash table*, where the corresponding value is to be inserted or sought. A *collision* arises when two different elements are mapped to the same array cell. In this paper we use *closed hashing*: If the cell where an element is to be found is occupied, one successively probes other cells until finding a free cell (insertions and unsuccessful searches) or until finding the element (successful searches).

We will consider two policies to determine the next cells to probe when a collision is detected at cell x. *Double hashing* computes another hash function y that depends on the key and probes $x + y$, $x + 2y$, etc. modulo the table size. *Linear probing* is a simpler policy. It tries the successive cells of the hash table, $x + 1$, $x + 2$, etc. modulo the table size.

The *load factor* is the fraction of occupied cells, and it influences space usage and time performance. Using good hash functions, insertions and unsuccessful searches require on average $1/(1 - \alpha)$ probes with double hashing, whereas successful searches require $\ln(1/(1 - \alpha))/\alpha$ probes. Linear probing requires more probes on average: $(1 + 1/(1 - \alpha)^2)/2$ for insertions and unsuccessful searches, and $(1 + 1/(1 - \alpha))/2$ for successful searches. Despite its poorer complexities, we consider also linear probing because it has advantages on some compressed representations we try.

Front-coding. Front-coding [33] is the folklore compression technique for lexicographically sorted dictionaries. It is based on the fact that consecutive entries are likely to share a common prefix. Each entry in the dictionary is be differentially encoded with respect to the preceding one. Two values are stored: an integer that encodes the length of their common prefix, and the remaining suffix of the current entry.

To allow searches, Front-Coding partitions the dictionary into buckets, where the first element is explicitly stored and the rest are differentially encoded. This allows the dictionary to be efficiently searched using a two-step process: first, a binary search on the first entry of the buckets locates the candidate bucket, and second a sequential scan of this candidate bucket rebuilds each element on the fly and compares it with the query. The bucket size yields a time/space tradeoff.

Front-coding has been sucessfully used in many applications. We emphasize its use in *WebGraph* (`http://webgraph.dsi.unimi.it`) to encode URL dictionaries from Web graphs.

Compressed text self-indexes. A compressed text *self-index* takes advantage of the compressibility of a text $T[1, N]$ to represent it in space close to that of the compressed text, while supporting random access and search operations. More precisely, a self-index supports at least operations $extract(i, j)$, which returns $T[i, j]$, and $locate(p)$, which returns the positions in T where pattern p occurs.

There are several self-indexes [28,11]. For this paper we are interested in particular in the *FM-index* family [12,13], which is based on the *Burrows-Wheeler transform (BWT)* [6]. FM-indexes achieve the best compression among self-indexes and are very fast to determine whether p occurs in T. Many self-indexes are implemented in the *PizzaChili* site (`http://pizzachili.dcc.uchile.cl`).

The BWT of $T[1,N]$, $T^{bwt}[1,N]$, is a permutation of its symbols. If the *suffixes* $T[i,N]$ of T are sorted lexicographically, then $T^{bwt}[j]$ is the character preceding the jth smallest suffix. We use the BWT properties in this paper to represent a dictionary as the FM-index of a text T.

FM-indexes support two basic operations on T^{bwt}. One is the *LF-step*, which moves from $T^{bwt}[j]$ that corresponds to the suffix $T[i,N]$ to $T^{bwt}[j']$ that corresponds to the suffix $T[i-1,N]$ (or $T[N,N]$ if $i=1$), that is $j' = LF(j)$. The second is the *backward step*, which moves from the lexicographical interval $T^{bwt}[sp,ep]$ of all the suffixes of T that start with string x to the interval $T^{bwt}[sp',ep']$ of all the suffixes that start with cx, for a character c.

Grammar-based compression. Grammar-based compresson is about finding a small grammar that generates a given text [7]. These methods exploit repetitions in the text to derive good grammar rules, so they are particularly suitable for texts containing many identical substrings. Finding the smallest grammar for a given text is NP-hard [7], so grammar-based compressors look for good heuristics. We use Re-Pair [23] as a concrete compressor, as it runs in linear time and yields good results in practice.

Re-Pair finds the most-repeated pair xy in the text and replaces all its ocurrences by a new symbol R. This adds a new rule $R \rightarrow xy$ to the grammar. The process iterates until all remaining pairs are unique in the text. Then Re-Pair outputs the set of r rules and the compressed text, $\mathcal{C}$. We use a public implementation (`http://www.dcc.uchile.cl/gnavarro/software`) for the compressor; each value (elements of a rule and symbols in $\mathcal{C}$) is stored in $\log(\sigma + r)$ bits.

Variable-length and direct-access codes. Brisaboa et al. [4] introduce a symbol reordering technique called directly addressable variable-length codes (*DACs*). Given a concatenated sequence of variable-length codes, DACs reorder the target symbols so that direct access to any code is possible. The overhead is at most one bit per target symbol, which is not too much if the target alphabet is large.

All the first symbols of the codes are concatenated in a first array A_1. A bitmap B_1 stores one bit per code in A_1, marking with a 1 the codes of length more than 1. The second symbols of the codes of length more than one are concatenated in a second array A_2, with B_2 marking which are longer than two, and so on. To extract the ith code, one finds its first symbol in $A_1[i]$. If $B_1[i]=0$, we are done. Otherwise we continue in $A_2[rank_1(B_1,i)]$, and so on.

A variable-length coding we use in this paper (albeit not in combination with DACs) is Vbyte [32]. It is used to represent numbers of distinct magnitudes, where most are small. Vbyte partitions the bits into 7-bit chunks and reserves the last bit of each byte to signal whether the number continues or not.

Tries and the XBW. A trie is an edge-labeled tree where each path from the root to a leaf represents a string. Strings that share a common prefix share a corresponding common path from the root.

A trie can represent a dictionary in a natural way. Searching for a string in the dictionary corresponds to following the labeled edges according to the string

characters. The number of the leaf would correspond to the *id* of the string (if the leaf exists), and it usually matches with the rank of that string in the set.

The main problem in practice is that tries tend to use much space; even when the space is linear, the constants are not negligible. To overcome this limitation, Ferragina et al. [15] proposed a compressed representation for trees that supports navigational operations, and subpath searching, using *rank* and *select* data structures for sequences. The representation, called *XBW*, corresponds to an extension of the BWT to trees.

3 Compressed Dictionary Representations

We describe now various approches for representing a dictionary within compressed space while solving two operations on it. The first operation, ***locate(p)***, gives a unique nonnegative identifier for the string p, if it appears in the dictionary; otherwise it returns -1. The second operation, ***extract(i)***, returns the string with identifier i in the dictionary, if it exists; otherwise returns $NULL$.

3.1 Hashing and Compression

We explore several combinations of hashing and compression. We Huffman-encode each string and the codes are concatenated in byte-aligned form. We insert the (byte-)offsets of the encoded strings in a hash table. The hash function operates over the encoded strings (seen as a sequence of bytes, that is, we compare them bytewise). This lowers the time to compute the function and to compare search keys (as the string is shorter). For searching we first Huffman-encode the search string and pad its bits to an integral number of bytes.

Our main hash function is a modified Bernstein's hash[1]. The second function for double hashing is the "rotating hash" proposed by Knuth [22, Sec. 6.4][2].

We concatenate the strings in the same order they are finally stored in the hash table. This improves locality of reference for linear probing, and gives other benefits, as seen later (in particular we easily know the length in bytes of each encoded string). We consider three variants to represent the hash table, and combine each of them with linear probing *(lp)* or double hashing *(dh)*.

The first variant, *Hash*, stores the hash table in classical form, as an array $H[1,m]$ pointing to the byte offset of the encoded strings. To answer *locate(p)* we proceed as usual, returning the offset of H where the answer was found, or -1 if not. To answer *extract(i)*, we simply decompress the string pointed from $H[i]$. Then with load factor $\alpha = n/m$ (n being the number of strings in the dictionary), the structure requires m integers in addition to the Huffman-compressed strings.

The second variant, *HashB*, stores $H[1,m]$ in compact form, that is, removing the empty cells, in an array $M[1,n]$. It also stores an *RG*-encoded bitmap $B[1,m]$

[1] `http://www.burtleburtle.net/bob/hash/doobs.html`. We initialize h as a large prime and replace the 33 by $2^{15}+1$, taking modulo the table size at each iteration.

[2] Precisely, the variant at `http://burtleburtle.net/bob/hash/examhash.html`. We also initialize h as a large prime.

that marks with a 1 the nonempty cells of H. Then $H[i]$ is empty if $B[i] = 0$, and if it is nonempty then its value is $H[i] = M[rank_1(B, i)]$. Now *locate(p)* returns positions in M, so our identifiers become contiguous in the range $[1, n]$, which is desirable. For *extract(i)* we simply decompress the string pointed from $M[i]$. The space of this representation is n integers plus $(1 + x)m$ bits, where x is the parameter of bitmap representation *RG*. The n integers require $n \log N$ bits, where N is the total byte length of the encoded strings.

The price is in time, as each new probe requires an additional *rank* on B. However, with linear probing, *rank* needs to be computed only once, as the successive cells are also successive in M. We only need to access the bits of B to determine where is the next empty cell.

The third variant, *HashBB*, also stores M and B instead of H, but M is replaced by a second bitmap. Note that since we have reordered the codes according to where they appear in H (or M), the values in these arrays are increasing. Thus instead of M we store a second bitmap $Y[1, N]$, where a 1 marks the beginning of the codes. Then $M[i] = select_1(Y, i)$. Bitmap Y is encoded in compressed form (*RRR*). Now the $n \log N$ bits of M are reduced to $\log \binom{N}{n} + (\frac{4}{15} + x)N$ bits, which is smaller unless the encoded strings are long.

The price is, again, in time. Each access to M requires a *select* operation. Note that linear probing does not save us from successive *select* operations, despite the involved string being contiguous, because we have no way to know where a code ends and the next starts.

3.2 Front-Coding and Compression

We consider two variants of Front-Coding. *Plain Front-Coding* implements the original technique by using Vbyte to encode the length of the common prefix. The remaining suffix is terminated with a zero-byte. Only bytewise operations are needed to search. The block sizes are measured in number of strings, so *extract(i)* determines the appropriate block with a simple division, and then scans the block to find the corresponding string.

Hu-Tucker Front-Coding is similar, but all the strings and Vbyte codes are encoded together using a single Hu-Tucker code. The bucket starts with the Hu-Tucker code of the first string, which is padded to the next byte boundary and preceded by the byte length of the encoded string, in Vbyte form. This prelude enables binary searching the first strings without decompressing them. The rest of the bucket is Hu-Tucker-compressed and bit-aligned, and is sequentially decompressed when scanning the bucket, both for locating and for extracting. We use a pointer-based Hu-Tucker tree implementation.

3.3 FM-Index Based Representation

We use two FM-indexes from *PizzaChili*. They represent the BWT using a *wavelet tree* [17], whose bitmaps are represented using *RG* (version `SSA_v3.1`) or *RRR* (version `SSA_RRR`). The former corresponds to the "succinct suffix array" [13], which achieves zero-order compression of T, and the second to the "implicit

compression boosting" idea [24], which reaches higher-order compression. Both FM-index implementations support functions LF and BWS, as well as obtaining $T^{bwt}[j]$ given j, in time $O(\log \sigma)$, where σ is the alphabet size of T. We use the indexes with no extra sampling because we need only limited functionality.

We concatenate all the strings in lexicographic order, terminating each one with a special character, \$, that is lexicographically smaller than all the symbols in T (in practice \$ is the ASCII code zero, which is the natural string terminator). We also add \$ at the beginning of the sequence. Thus we can speak of the ith string in lexicographical or positional order, indistinctly.

Note that, when the suffixes of T are sorted lexicographically, the first corresponds to the final \$, and the next n correspond to the \$s that precede each dictionary string. Thus $T^{bwt}[1]$ is the final character of the nth dictionary string, and $T^{bwt}[i+2]$ is the final character of the ith string, for $1 \leq i < n$. Therefore $extract(i)$ can be carried out by starting at the corresponding position of T^{bwt} and using LF-steps until reaching a \$. The $T^{bwt}[j]$ characters traversed spell out the desired dictionary string in reverse order.

To answer *locate(p)* we just need to determine whether $\$p\$$ occurs in T. Thus we start with $(sp, ep) = (1, n+1)$ and use $|p|+1$ backward steps until finding the lexicographical interval (sp', ep') of the suffixes that start with $\$p\$$. If p exists in the dictionary and is the ith string, then $sp' = ep' = i+1$ and we simply return i; otherwise $sp' > ep'$ holds at some point in the process and we return -1.

3.4 Re-pair Based Representation

We concatenate all the dictionary strings in lexicographic order and apply Re-Pair compression to the concatenation. However, we avoid forming rules that contain the string terminator. This ensures that each string is encoded with an integral number of symbols in $\mathcal{C}$ and thus decompression is fast.

Locating is done via binary search, where each dictionary string to compare must be decompressed first. We decompress the string only up to the point where the lexicographical comparison can be decided. For extraction we simply decompress the desired string.

For both operations we need direct access to the first symbol of the ith string in $\mathcal{C}$. Each compressed string can be seen as a variable-length sequence of symbols in $\mathcal{C}$, where they are concatenated. Thus we use the DAC representation on those sequences. This gives fast direct access to the ith string, at the price of 1.25 bits per symbol: we use RG representation with 25% overhead.

3.5 XBW Trie Representation

If the trie has N nodes, the XBW consists of a sequence $S_\alpha[1, N+n]$ of labels (each leaf is identified with a label \$ leading to it) plus a bitmap $S_{last}[1, N+n]$ with n bits set. We represent S_α using wavelet trees and, as for the FM-Index, represent their bitmaps (and S_{last}) using RG or RRR.

For locating, we use operation `GetChildren` [15] to find the leaf. Then we map the leaf x to an identifier in the range $[1, n]$ with $rank_\$(S, x)$. For extracting, we start from the leaf and use `GetParent` [15] to obtain all the string characters.

4 Experimental Results

We consider four dictionaries that are representative of relevant applications:

Words comprises all the different words with at least 3 ocurrences in the ClueWeb09 dataset It contains 25,609,784 words and occupies 256.36 MB.
DNA all substrings of 12 nucleotides found in S. Paradoxus, known as the para dataset[3]. It contains 9,202,863 subsequences and occupies 114.09 MB.
URLs corresponds to a 2002 crawl of the .uk domain from the *WebGraph* framework. It contains 18,520,486 URLs and occupies 1.34 GB.
URIs contains all different URIs used in the *DBpedia-en* RDF dataset (blank nodes and literals excluded). It contains 30,176,012 URIs and takes 1.52 GB.

We use an Intel Core2 Duo processor at 3.16 GHz, with 8 GB of main memory and 6 MB of cache, running Linux kernel 2:6:24-28. We ran *locate* experiments for successful and unsuccessful searches. For the former we chose 10,000 dictionary strings at random. For the latter we chose other 1,000 strings at random and excluded them from the indexing. For *extract* we queried 10,000 random numbers between 1 and n. Each data point is the average user time over 10 repetitions.

Figure 1 shows our results. Most methods are drawn as a line that corresponds to their main space/time tuning parameter. On the left we show locate time for successful searches; the plots for unsuccessful searches are very similar and omitted for lack of space. On the right we show extraction times. Time is shown in microseconds and space as a percentage of the space required by concatenating the plain strings. Since, despite the advantages of linear probing in this scenario, double hashing was always better, we only plot the latter.

Front-Coding with Hu-Tucker compression shows to be an excellent choice in all cases, achieving good time performance and the least space usage (only beaten by *XBW* and, on URLs, by *Re-Pair*). The folklore *Front-Coding*, without compression, is almost everywhere dominated by the compressed variant.

The least space is always achieved by *XBW+RRR*, yet the time it achieves is significantly higher than the other approaches. The next best space, on URLs, is achieved by *Re-Pair*, which is much faster than *XBW* but still noticeably slower than compressed *Front-Coding*. On the shorter-string dictionaries (Words and DNA), *Re-Pair* does not compress well and compressed *Front-Coding* achieves the second-best space (with much better time than *XBW* variants).

HashBB performs better in space than *HashB* when the strings are short, otherwise the bitmap becomes too long. It is never, however, clearly the best alternative. *HashB* and *Hash* excell in time with short strings when much space is used (nearly 100%), yet *HashB* is never much better than *Hash*.

For extracting, the map is dominated by *Front-Coding*, in plain or compressed form (the plain folklore variant is more relevant in this case). Still *Re-Pair* achieves less space on URLs, and *XBW* always requires the minimum space but the highest times.

[3] http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp

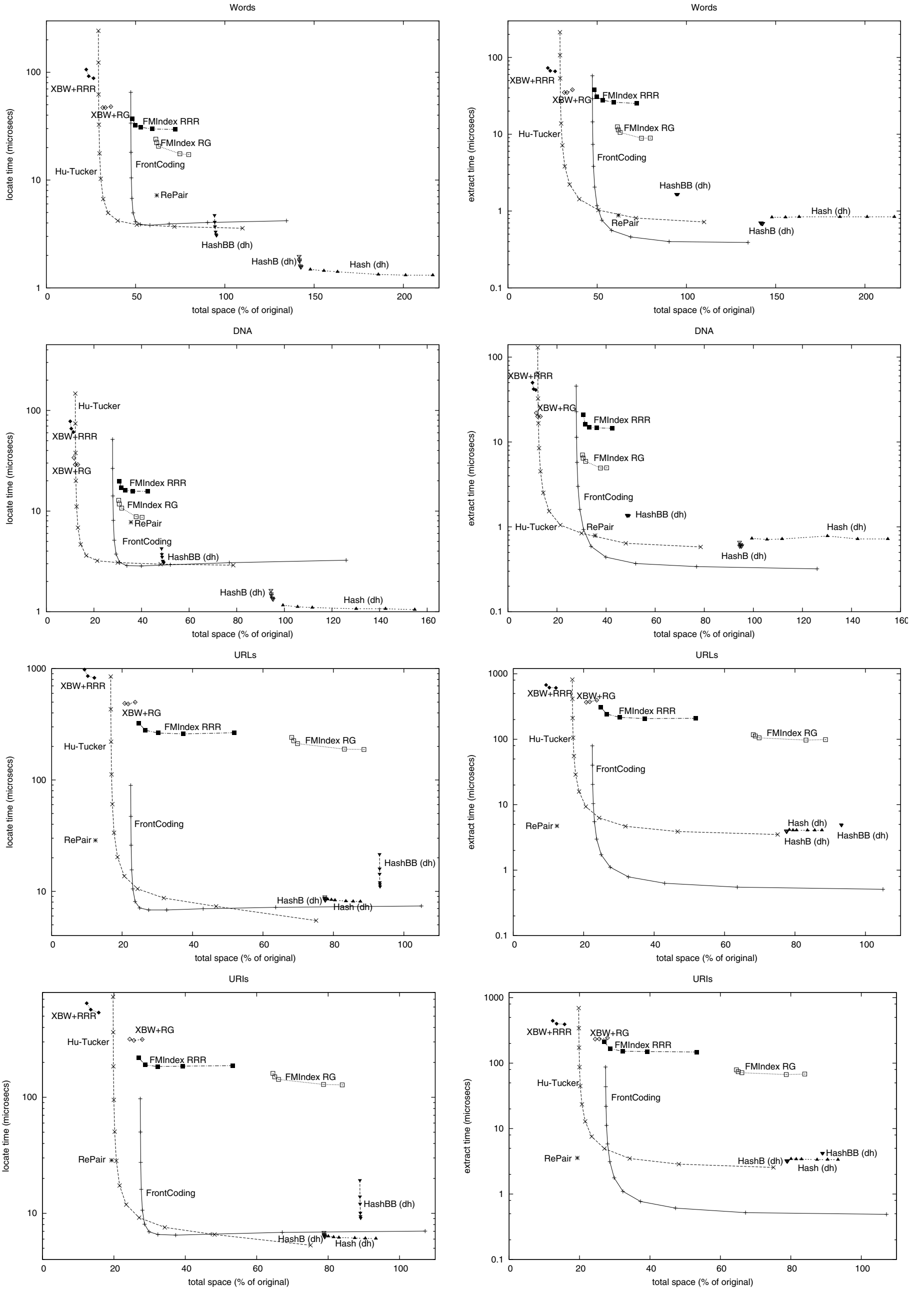


Fig. 1. Locate times (left) and extract times (right) for the different methods as a function of their space consumption

5 Final Remarks

Prefix search, that is, finding the dictionary strings that start with a given pattern, is easily supported by the methods we have explored, except hashing. Other variants that can likewise be supported are of interest for Internet routing tables, e.g., find the dictionary string that is the longest prefix of the pattern.

Despite the *FM-index* and the *XBW* being the slowest solutions, they support other searches of interest, such as finding the dictionary strings that contain a substring, or that have a given prefix and a given suffix [14,15]. They also support approximate searches [30].

We have reordered the strings at our convenience, but sometimes the order must be fixed. Hashing is easily adapted to any order (except the variant *HashBB*), but others would need an explicit permutation that would significantly increase the space. The *FM-index* and the *XBW* can use the LF-step mechanism to trade space for time and store just a sample permutation.

References

1. Apostolico, A., Drovandi, G.: Graph compression by BFS. Algorithms 2, 1031–1044 (2009)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison Wesley, Reading (1999)
3. Boldi, P., Vigna, S.: The Webgraph framework i: Compression techniques. In: Proc. WWW, pp. 595–602 (2004)
4. Brisaboa, N., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 122–130. Springer, Heidelberg (2009)
5. Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.: Graph structure in the Web. Comput. Netw. 33, 309–320 (2000)
6. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation (1994)
7. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theory 51(7), 2554–2576 (2005)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill (2001)
9. Donato, D., Laura, L., Leonardi, S., Meyer, U., Millozzi, S., Sibeyn, J.: Algorithms and experiments for the Webgraph. J. Graph Algor. App. 10(2), 219–236 (2006)
10. Fernández, J.D., Martínez-Prieto, M.A., Gutierrez, C.: Compact representation of large RDF data sets for publishing and exchange. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 193–208. Springer, Heidelberg (2010)
11. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM JEA 13, article 12 (2009)
12. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. FOCS, pp. 390–398 (2000)

13. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), article 20 (2007)
14. Ferragina, P., Venturini, R.: The compressed permuterm index. ACM Trans. Alg. 7(1), article 10 (2010)
15. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proc. FOCS, pp. 184–196 (2005)
16. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. Posters WEA, pp. 27–38 (2005)
17. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. SODA, pp. 841–850 (2003)
18. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge Univ. Press, Cambridge (2007)
19. Heaps, H.S.: Information Retrieval: Computational and Theoretical Aspects. Academic Press, London (1978)
20. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. of the Institute of Radio Engineers 40(9), 1098–1101 (1952)
21. Kleinberg, J., Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.: The Web as a graph: Measurements, models, and methods. In: Asano, T., Imai, H., Lee, D.T., Nakano, S.-i., Tokuyama, T. (eds.) COCOON 1999. LNCS, vol. 1627, pp. 1–17. Springer, Heidelberg (1999)
22. Knuth, D.E.: The Art of Computer Programming, volume 3: Sorting and Searching. Addison Wesley, Reading (2007)
23. Larsson, N.J., Moffat, J.A.: Offline dictionary-based compression. Proc. of the IEEE 88, 1722–1732 (2000)
24. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 229–241. Springer, Heidelberg (2007)
25. Manning, C.D., Schütze, H.: Foundations of Statistical Natural Language Processing. MIT Press, Cambridge (1999)
26. Moffat, A., Katajainen, J.: In-place calculation of minimum-redundancy codes. In: Sack, J.-R., Akl, S.G., Dehne, F., Santoro, N. (eds.) WADS 1995. LNCS, vol. 955, pp. 393–402. Springer, Heidelberg (1995)
27. Nagwani, N.: Clustering based URL normalization technique for Web mining. In: Proc. ACE, pp. 349–351 (2010)
28. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
29. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. SODA, pp. 233–242 (2002)
30. Russo, L., Navarro, G., Oliveira, A., Morales, P.: Approximate string matching with compressed indexes. Algorithms 2(3), 1105–1136 (2009)
31. Suel, T., Yuan, J.: Compressing the graph structure of the Web. In: Proc. DCC, pp. 213–222 (2001)
32. Williams, H., Zobel, J.: Compressing integers for fast file access. The Computer Journal 42, 193–201 (1999)
33. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, San Francisco (1999)
34. Yin, M., Goh, D., Lim, E.-P., Sun, A.: Discovery of concept entities from Web sites using web unit mining. Intl. J. of Web Inf. Sys. 1(3), 123–135 (2005)

Combinatorial Optimization for Weighing Matrices with the Ordering Messy Genetic Algorithm

Christos Koukouvinos and Dimitris E. Simos

Department of Mathematics, National Technical University of Athens,
Zografou 15773, Athens, Greece
{ckoukouv,dsimos}@math.ntua.gr

Abstract. In this paper, we demonstrate that the search for weighing matrices constructed from two circulants can be viewed as a permutation problem. To solve it a set of two competent genetic algorithms (CGAs) are used to locate common integers in two sorted arrays. The motivation to deal with the messy genetic algorithm (mGA) is given from the pioneering results of Goldberg, regarding the ability of the mGA to put tight genes together in a solution which points directly to structural patterns in weighing matrices. In order to take into advantage a recent formalism on the support of two sequences with zero autocorrelation we use an adaptation of the ordering messy GA (OmeGA) where we combine the fast mGA with random keys to represent permutations of the two sequences under investigation. This transformation of the weighing matrices problem to an instance of a combinatorial optimization problem seems to be promising since we illustrate that our framework is capable to solve open cases for weighing matrices as these are listed in the second edition of the Handbook of Combinatorial Designs.

Keywords: Weighing matrices, messy genetic algorithm, ordering messy genetic algorithm, competent metaheuristics, optimization.

1 Introduction

A square $n \times n$ matrix with elements from $\{-1, 0, 1\}$ such that $WW^T = wI_n$, where W^T stands for the transpose matrix of W, will be called a weighing matrix of order n and weight w, denoted by $W(n, w)$. Authoritative information for weighing matrices can be found in [23] and [24].

In this paper we focus our attention on weighing matrices constructed from two circulants. The following "plug-in" method for constructing weighing matrices is described in the Theorem below, see [4].

Theorem 1. *If there exist two circulant matrices A, B of order n, with entries from $\{0, \pm 1\}$, satisfying $AA^T + BB^T = wI_n$ and w is an integer, then there exists a $W(2n, w)$, given as*

$$W(2n, w) = \begin{pmatrix} A & B \\ -B^T & A^T \end{pmatrix} \quad \text{or} \quad W(2n, w) = \begin{pmatrix} A & BR \\ -BR & A \end{pmatrix}$$

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 148–156, 2011.
© Springer-Verlag Berlin Heidelberg 2011

where R is the square matrix of order n with $r_{ij} = 1$ if $i + j - 1 = n$ and 0 otherwise.

We adopt the following definitions from [22].

Definition 1. *Let $A = [a_1, a_2, \ldots, a_n]$ be a sequence of length n. The periodic autocorrelation function, PAF of A, $PAF_A(s)$ is defined as:*

$$PAF_A(s) = \sum_{i=1}^{n} a_i a_{i+s} \ , \ s = 0, 1, \ldots, n-1.$$

where $i + s$ is taken modulo n, when $i + s > n$.

Definition 2. *Two sequences, $A = [a_1, \cdots, a_n]$ and $B = [b_1, \cdots, b_n]$, of length n are said to have zero PAF, if $PAF_A(s) + PAF_B(s) = 0$ for $s = 1, \ldots, n-1$.*

In order to find suitable circulant submatrices that satisfy the additive property in Theorem 1 we use an important result that comes from sequences with zero periodic autocorrelation function (PAF), as outlined in [24].

Remark 1. If there are two sequences A and B of length n with entries from $\{0, \pm 1\}$ and total weight w with zero periodic autocorrelation function, then these sequences can be used as the first rows of circulant matrices which can be used in the construction of Theorem 1 to form a weighing matrix of order $2n$ and weight w.

These double circulant sequences will be denoted by $DC(n, w)$ if they have zero PAF, i.e. if they satisfy the zero PAF condition given in Definition 2. Sequences of the previous form, are also called periodic complementary sequences.

1.1 Applications of Periodic Complementary Sequences

$DC(n, w)$ are used to construct sequences with desirable properties for radar applications, as described in [30]. Moreover, these sequences intervene in coded aperture imaging ([3]) and higher-dimensional signal processing applications such as time-frequency-coding ([5]) or spatial correlation, [10]. Sequences of the above type are also of greatest importance, when constructing weighing matrices whose significance has exhibited in other fields, such as quantum information processing [2].

Last we would like to mention that such sequences, are interesting objects to study for themselves [24], [29] .

2 Recent Progress for Searching Weighing Matrices

Lemma 11 of [24] describes in a compact form the progress made till 1999, for searching weighing matrices constructed from two circulants. A necessary condition for the existence of $W(2n, w)$ constructed from two circulants, dictates that the Diophantine equation $a^2 + b^2 = w$ has solutions. Hence, we keep focus

on the permissible odd values of n, i.e. values of n such that the Diophantine equation $a^2 + b^2 = w$ has solutions.

Computational optimization algorithms and techniques for searching weighing matrices has recently been studied in [15],[16],[17],[18],[19]. In these works, the authors restrained their interest for weighing matrices of large weight.

The pursuit of resolving unknown cases of $W(2n, 2n - \alpha)$ weighing matrices constructed from two circulants, where the weight $w = 2n - \alpha$ is expressed as a function of the order n, via discovering structural patterns for the location of the α zeros in the two arrays $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$ has met a recent boost by several researchers. Important results of this technique can be found in [15],[16],[17]. The problem of searching for weighing matrices was also phrased as a Combinatorial Optimization problem, as shown in [18],[19].

In this paper, we follow our approach given in [18], where we expressed the structural patterns through linkage learning techniques in order to employ competent genetic algorithms and to construct a number of new weighing matrices constructed from two circulants. In particular, we enhance our prior adaptation of fast messy genetic algorithm (fmGA) [18], by employing random keys for representing chromosomes. This technique has the advantage that the fmGA is transformed easily to a permutation solving GA, the so called ordering messy GA (OmeGA), [14]. To the best of our knowledge this is the first time that this variant of fmGA is applied successfully in the search for weighing matrices.

3 Messy Genetic Algorithms for Weighing Matrices

Since the pioneering results of Holland and De Jong on genetic algorithms, a lot of researchers have taken serious effort to design sophisticated (competent) algorithms for combinatorial optimization problems, see for example [11] and [12]. For a general overview of evolutionary algorithms, we refer to [26],[27]. The motivation to originally deal with the messy genetic algorithm (mGA) was given from the pioneering results of Goldberg [6], regarding the ability of the mGA to put tight genes together in a solution, e.g. $(0000 * *)$. This ability of the mGA points directly to the structural patterns mentioned earlier, which in our framework was explored from a combinatorial optimization point of view.

Though, the implementation details and the formulation of the weighing matrices problem in a encoding suitable for mGA appeared in [18], we shall try to outline the basic concepts of our algorithm in a compact form. We used an adaptation of an improved version of the mGA, the fast mGA, as given in [8] in order to avoid initialization bottlenecks.

Recall that the permissible entries for the two circulant submatrices, that form a weighing matrix, arise from the set $\{-1, 0, 1\}$. Representing genes as ordinary integer values, genetic algorithms for combinatorial problems typically utilize an integer encoding for the chromosomes. Therefore, we have used GA operators that have been developed to maintain feasibility in terms of gene duplication in the population when using integer encoding [28].

The major advantage of the mGA is to consider solutions of variable length. For example, in the following messy encoding we manufactured, the solutions

$((1,1,0),(2,1,-1),(2,2,0))$ and $((1,1,1),(1,2,1),(2,1,1),(2,2,-1),(2,2,0))$ are both valid for a 4-bit problem suitable to search for weighing matrices of order 4. The first solution is decoded as $[0,*]$ and $[-1,0]$, since the encoding $(1,1,0)$ simply means in the first sequence, the first entry is zero. In particular, the messy genes are represented by tuples which define their position (locus) and value (allele). We used an additional index to clarify the sequence used,

$$\text{messy gene } g : (sequence\ index, position, value) \tag{1}$$

There must be no confusion for the underspecification in the first solution (no 2nd bit in the first sequence) and the overspecification in the latter (two different 2nd bits in the second sequence).

Goldberg [6], proposes a gene expression operator that employs a first-come-first-served rule on a left-to-right scan to handle overspecification. Thus, in the second solution this left-to-right scan drops the second instance $(2,2,0)$, obtaining the valid two sequences $[1,1]$ and $[1,-1]$ that in the sequel form the weighing matrix. In the case of underspecification, the unspecified genes are filled in using a competitive template, which is a fully specified chromosome from which any missing genes are directly inherited. For example, using as competitive template the messy encoding $((1,1,1),(1,1,-1),(2,1,-1),(2,2,0))$ the first solution inherit a -1 in the 2nd bit of its first sequence, i.e. $[0,-1]$ and $[-1,0]$. Obviously, genes that are already specified in the solution do not take into account the competitive template.

This case, is of particular interest since it provides us with the ability to guide the algorithm to a particular partition of the solution space. We have chosen to use as competitive templates, structural patterns that have proven to be successful in the past, thus providing our algorithm with a linkage learning technique similar to the one explored in [7]. Division of evolutionary processing in the mGA comes in two phases: primordial and juxtapositional. For an overview of these two phases we refer to [6],[7]. Allowing variable-length chromosomes, overspecified, or underspecified solutions means that the usual simple crossover operator used will no longer work. In the context of mGA the crossover is replaced with two simpler operators, splice and cut which we have used as they are described in [6].

For example, starting with the two solutions $((1,1,0),(2,2,1),(1,2,-1))$ and $((1,1,1),(2,1,0))$ splice operator would yield the single solution $((1,1,0),(2,2,1),$ $(1,2,-1),(1,1,1),(2,1,0))$. Also applied the cut operator to the solution $((1,1,0),$ $(2,2,1),(1,2,-1),(1,1,1),(2,1,0))$; supposing that a cut at location 2 was indicated, the two solutions $((1,1,0),(2,2,1))$ and $((1,2,-1),(1,1,1),(2,1,0))$ would be obtained.

Finally, we enriched the fast messy GA with two techniques, thresholding and tie-breaking, to overcome the problem of cross-competition of common messy genes and to successfully address the problem of non uniform building block (BB) size that occurred in some cases, respectively.

4 An Added Level of Sophistication for Searching Weighing Matrices: OmeGA

In the previous Section, we have presented the formulation of the fmGA for weighing matrices assuming that it operates on ternary strings corresponding to the ternary DC pairs. To specialize the algorithm for a new formalism for sequences with zero (periodic) autocorrelation function [25] we have to choose a suitable representation.

The ordering messy genetic algorithm (OmeGA) [14] is a fast messy genetic algorithm (fmGA), specialized for permutation problems. It represents the chromosomes by vectors of real numbers, the so-called *random keys* introduced by Bean [1]. In a number of experiments it is shown that OmeGA significantly outperforms the simple GA in solving ordering deceptive problems [13].

4.1 Design of the OmeGA

This Section overviews a new formalism for the computation of the PAF of a $DC(n, w)$ [25], and explains the concept of random keys and the random key-based simple GA (RKGA), [1]. An important variant of the later algorithm is the biased random key-based simple GA (BRKGA), [9]. The success of our proposed framework for OmeGA is based in the following rules:

- All mechanisms of the fmGA are applied
- The alleles are (long) integer numbers
- The alleles are treated as random keys to encode permutations

Multisets for sequences with zero PAF can be naturally defined by using the formalism given in [25] for sequences with zero NPAF, using the symmetric relation, $PAF_A(s) = NPAF_A(s) + NPAF_A(n-s),\ s = 1, 2, \ldots, n-1$ where the non-periodic autocorrelation function (NPAF) of a sequence $A = [a_1, a_2, \ldots, a_n]$ of length n is defined as $N_A(s) = \sum_{i=1}^{n-s} a_i a_{i+s}$, $s = 0, 1, \ldots, n-1$. We formally define the positive and negative support of the sequence A as $POS(A) = \{i : a_i > 0 \mid i = 1, \ldots, n\}$ and $NEG(A) = \{j : a_j < 0 \mid j = 1, \ldots, n\}$. The main idea is to work with the support of a sequence and bundle together the indices of entries with the same sign.

Following [31] we are concerned with multisets denoted by square brackets ([ ]), defined on the fixed group $\mathbb{Z}_n$ of order n, in which repeated elements are counted multiply. If T_1 and T_2 are two lists then by $T_1 \uplus T_2$ we denote the result of appending the elements of T_1 to T_2 (with multiplicities retained). If the resulting list is sorted after appending, the operation is denoted by $T_1 \& T_2$. We define the occurrences counting function $[T]_e$ for a multiset T and an element from the domain of elements of S as $[T]_e = |\{x \in T | x = e\}|$. For example, let T be the multiset $T = [1, 1, 2, 2, 2, 4]$; then $[T]_1 = 2, [T]_2 = 3, [T]_3 = 0$ and $[T]_4 = 1$. It follows that $[T_1 \uplus T_2]_e = [T_1]_e + [T_2]_e$. For prior usage of multisets in the study of sequences with zero autocorrelation function we refer to [25],[31], while for related operations on them see [21].

Then we define the signed and cross-differences as $D^{+}_{A,1} = [x - y : x > y \text{ and } x, y \in POS(A)]$, $D^{-}_{A,1} = [x - y : x > y \text{ and } x, y \in NEG(A)]$ and $D^{\pm}_{A,1} = [x - y : x > y \text{ and } x \in POS(A), y \in NEG(A)]$, $D^{\mp}_{A,1} = [x - y : x > y \text{ and } x \in NEG(A), y \in POS(A)]$, respectively. Also we define $C^{\rightrightarrows}_{A,1} = D^{\pm}_{A,1} \uplus D^{\mp}_{A,1}$. Then we can prove the following lemma which acts as a criterion to decide if two sequences form a $DC(n, w)$.

Lemma 1. *Let A, B be two sequences of length n and weight w with entries from $\{0, \pm 1\}$. Let also D be $(D^{+}_{A,2} \uplus D^{-}_{A,2}) \uplus (D^{+}_{B,2} \uplus D^{-}_{B,2})$ and C be $C^{\rightleftarrows}_{A,2} \uplus C^{\rightleftarrows}_{B,2}$. Then, the following are equivalent:*

(i) A,B is $DC(n, w)$
(ii) $[D]_s = [C]_s$ for $s \in \{1, 2, \ldots n - 1\}$

where, $D^{+}_{A,2} = \{D^{+}_{A,1}, D^{+}_{A,1} \pmod n\}$, $D^{-}_{A,2} = \{D^{-}_{A,1}, D^{-}_{A,1} \pmod n\}$ and $C^{\rightleftarrows}_{A,2} = \{C^{\rightrightarrows}_{A,1}, C^{\rightrightarrows}_{A,1} \pmod n\}$.

Proof. We have that A,B form a $DC(n, w) \Leftrightarrow PAF_A(s) + PAF_B(s) = 0,\ s = 1, \ldots n - 1$. We are interested in finding how many pairs of (a_i, a_{i+s}) in the support have distance s and how many such pairs will result to a positive or negative value in the $PAF_A(s) + PAF_B(s)$, for a fixed s. There is a distinct number of such cases that can contribute a "1" or "−1" in the $PAF_A(s) + PAF_B(s)$. In particular, when $s \in D^{+}_{A,2} \cup D^{+}_{B,2}$ or $s \in D^{-}_{A,2} \cup D^{-}_{B,2}$ a "1" is contributed to the $PAF_A(s) + PAF_B(s)$ since $a_i a_{i+s} = 1$ for $a_i = a_{i+s} = 1$ or $a_i = a_{i+s} = -1$, respectively. Moreover, when $s \in C^{\rightleftarrows}_{A,2} \cup C^{\rightleftarrows}_{B,2}$ a "−1" is contributed to the $PAF_A(s) + PAF_B(s)$, since $a_i a_{i+s} = -1$. Formally, for $s \in \{1, 2, \ldots n - 1\}$ we have:

$$\begin{aligned}
PAF_A(s) + PAF_B(s) &= ([D^{+}_{A,2}]_s + [D^{-}_{A,2}]_s - [C^{\rightleftarrows}_{A,2}]_s) + ([D^{+}_{B,2}]_s + [D^{-}_{B,2}]_s - [C^{\rightleftarrows}_{B,2}]_s) \\
&= ([D^{+}_{A,2}]_s + [D^{-}_{A,2}]_s + [D^{+}_{B,2}]_s + [D^{-}_{B,2}]_s) - ([C^{\rightleftarrows}_{A,2}]_s + [C^{\rightleftarrows}_{B,2}]_s) \\
&= [D^{+}_{A,2} \uplus D^{-}_{A,2}]_s + [D^{+}_{B,2} \uplus D^{-}_{B,2}]_s - [C^{\rightleftarrows}_{A,2} \uplus C^{\rightleftarrows}_{B,2}]_s \\
&= [D]_s - [C]_s
\end{aligned}$$

Thus

$$PAF_A(s) + PAF_B(s) = 0 \Leftrightarrow [D]_s - [C]_s = 0 \Leftrightarrow [D]_s = [C]_s. \qquad \square$$

Remark 2. For an immediate validation of Lemma 1 in terms of a computer implementation, it is more convenient to consider the resulting lists C and D to be sorted, i.e. $D = (D^{+}_{A,2} \uplus D^{-}_{A,2}) \& (D^{+}_{B,2} \uplus D^{-}_{B,2})$ and $C = C^{\rightleftarrows}_{A,2} \& C^{\rightleftarrows}_{B,2}$.

Using random keys for representation as outlined in [1] enable us to use (long) integer numbers, corresponding to the support of the sequences under investigation, as sort keys to decode these sequences. Hence, we achieve a more compact description of the support of a sequence. We represent a permutation

of length ℓ as an integer vector $\mathbf{r} = (r_1, r_2, \ldots, r_\ell)$ where $\mathbf{r} \in [-n, n]^\ell$. By sorting the random keys in

$$\text{ordering messy gene } g : (\textit{sequence index}, \textit{position}, \textit{random key}) \tag{2}$$

such that

$$r_{\phi(1)} \leq r_{\phi(2)} \leq \ldots \leq r_{\phi(\ell)}$$

holds, where $\phi : \{1, \ldots, \ell\} \longrightarrow \{1, \ldots, \ell\}$ is the corresponding mapping function arranging the keys in ascending order, the permutation is decoded as follows:

$$(\phi(1), \phi(2), \ldots, \phi(\ell))$$

For the weighing matrices problem we have that $\ell = w$ since the integers represent the support of the candidate $DC(n, w)$.

The choice of the objective function (OF) arises naturally as the minimum number of random keys that have to be changed to transform one permutation into another. Clearly, when this value is equal to zero we have that the candidate sequences form a $DC(n, w)$ due to the computation of the PAF expressed by signed difference sets as outlined in Lemma 1 via Remark 2. Caution is needed not to confuse that the support of a sequence is a set, whilst the computation of the PAF of a sequence (expressed by its support) is a multiset.

The first results of the execution of the OmeGA for searching weighing matrices seems to be promising, and we present here the following $DC(61, 72)$ which can be used to form a $W(122, 72)$ in Theorem 1, which is listed as open in Table 6 of [24].

```
--000+00--0+---+---+--0+000-++000000---0++00+00-+-++-+-0+000-
--000-00--0---+-+++-++0-000+--000000++-0--00+00-+-++-+-0+000-
```

Acknowledgments

The authors are thankful to the anonymous reviewers for their useful comments and suggestions that lead to an improvement of the presentation of the paper. The research of the second author was financially supported by a scholarship awarded by the Secretariat of the Research Committee of National Technical University of Athens.

References

1. Bean, J.C.: Genetic algorithms and random keys for sequencing and optimization. ORSA J. on Computing 6, 154–160 (1994)
2. van Dam, W.: Quantum algorithms for weighing matrices and quadratic residues. Algorithmica 34, 413–428 (2002)
3. Fenimore, E., Cannon, T.: Coded aperture imaging with uniformly redundant array. Appl. Optics 17, 337–347 (1978)

4. Geramita, A.V., Seberry, J.: Orthogonal Designs. Quadratic Forms and Hadamard Matrices. Lecture Notes in Pure and Applied Mathematics, vol. 45. Marcel Dekker, Inc., New York (1979)
5. Golomb, S., Taylor, H.: Two-dimensional synchronization patterns for minimum ambiguity. IEEE Trans. Inform. Theory 28, 600–604 (1982)
6. Goldberg, D.E., Deb, K., Korb, B.: Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems 5, 493–530 (1989)
7. Goldberg, D.E., Deb, K., Korb, B.: Messy genetic algorithms revisited: Studies in mixed size and scale. Complex Systems 4, 415–444 (1990)
8. Goldberg, D.E., Deb, K., Kargupta, H., Harik, G.: Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms. In: Proceedings of the Fifth International Conference on Genetic Algorithms, pp. 56–64. Morgan Kaufmann Publishers Inc., San Francisco (1993)
9. Goncalves, J.F., Resende, M.G.C.: Biased random-key genetic algorithms for combinatorial optimization (to appear in Journal of Heuristics)
10. Hersheya, J., Yarlagadda, R.: Two-dimensional synchronisation. Electron. Lett. 19, 801–803 (1983)
11. Holland, J.H.: Adaptation in Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. University of Michigan Press, Ann Arbor (1975)
12. De Jong, K.A.: An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Doctoral Thesis, CCS Department. University of Michigan, Ann Arbor, MI (1975)
13. Kargupta, H., Deb, K., Goldberg, D.E.: Ordering Genetic Algorithms and Deception. In: Männer, R., Manderick, B. (eds.) Parallel Problem Solving from Nature PPSN II, pp. 47–56. Elsevier Science Publishers B.V. (1992)
14. Knjazew, D.: OmeGA: A Competent Genetic Algorithm for Solving Permutation and Scheduling Problems. Kluwer, Norwell (2002)
15. Kotsireas, I.S., Koukouvinos, C., Seberry, J.: Weighing matrices and string sorting. Annals of Combinatorics 13, 305–313 (2009)
16. Kotsireas, I.S., Koukouvinos, C., Pardalos, P.M.: An efficient string sorting algorithm for weighing matrices of small weight. Optimization Letters 4, 29–36 (2010)
17. Kotsireas, I.S., Koukouvinos, C., Pardalos, P.M.: A modified power spectral density test applied to weighing matrices with small weight (to appear in J. Comb. Optim.)
18. Kotsireas, I.S., Koukouvinos, C., Pardalos, P.M., Simos, D.E.: Competent genetic algorithms for weighing matrices (submitted for publication)
19. Kotsireas, I.S., Koukouvinos, C., Pardalos, P.M., Shylo, O.: Periodic complementary binary sequences and combinatorial optimization algorithms. J. Comb. Optim. 20, 63–75 (2010)
20. Kharaghani, H., Koukouvinos, C.: Complementary, Base and Turyn Sequences. In: Colbourn, C.J., Dinitz, J.H. (eds.) Handbook of Combinatorial Designs, 2nd edn., pp. 317–321. Chapman and Hall/CRC Press, Boca Raton, Fla (2006)
21. Knuth, D.E.: The Art of Computer Programming, 3rd edn. Seminumerical Algorithms of Addison- Wesley Series in Computer Science and Information Processing, vol. 2. Addison-Wesley Publishing Co., Mass. (1998)
22. Koukouvinos, C.: Sequences with Zero Autocorrelation. In: Colbourn, C.J., Dinitz, J.H. (eds.) The CRC Handbook of Combinatorial Designs, pp. 452–456. CRC Press, Boca Raton (1996)
23. Koukouvinos, C., Seberry, J.: Weighing matrices and their applications. J. Statist. Plann. Inference 62, 91–101 (1997)

24. Koukouvinos, C., Seberry, J.: New weighing matrices and orthogonal designs constructed using two sequences with zero autocorrelation function - a review. J. Statist. Plann. Inference 81, 153–182 (1999)
25. Koukouvinos, C., Simos, D.E.: On the computation of the non-periodic autocorrelation function of two ternary sequences and its related complexity analysis (to appear in J. Appl. Math. & Informatics)
26. Pardalos, P.M., Du, D.-Z. (eds.): Handbook of Combinatorial Optimization. Combinatorial Optimization, vol. 2. Kluwer Academic Publishers, Springer Netherlands (1998)
27. Pardalos, P.M., Resende, M.G.C. (eds.): Handbook of Applied Optimization. Oxford University Press, Inc., 198 Madison Avenue, USA (2002)
28. Rothlauf, F.: Representations for Genetic and Evolutionary Algorithms, 2nd edn. Physica-Verlag, Heidelberg (2006)
29. Seberry, J., Yamada, M.: Hadamard Matrices, Sequences and Block Designs. In: Dinitz, J.H., Stinson, D.R. (eds.) Contemporary Design Theory: A Collection of Surveys, pp. 431–560. John Wiley & Sons, New York (1992)
30. Weathers, G., Holiday, E.M.: Group-complementary array coding for radar clutter rejection. IEEE Transaction on Aerospace and Electronic Systems 19, 369–379 (1983)
31. Wallis, J.S.: On supplementary difference sets. Aequationes Math. 8, 242–257 (1972)

Improved Automated Reaction Mapping

Tina Kouri and Dinesh Mehta*

Colorado School of Mines
{tkouri,dmehta}@mines.edu

Abstract. Automated reaction mapping is an important tool in cheminformatics where it may be used to classify reactions or validate reaction mechanisms. The reaction mapping problem is known to be NP-Complete and may be formulated as an optimization problem. In this paper we present three algorithms that continue to obtain optimal solutions to this problem, but with significantly improved runtimes over the previous CCV algorithm. Our algorithmic improvements include (a) the use of a fast (but not 100% accurate) canonical labeling algorithm, (b) name reuse (i.e., storing intermediate results rather than recomputing), and (c) an incremental approach to canonical name computation. Experimental results on chemical reaction databases demonstrate our 2-CCV NR FDN algorithm usually performs over ten times faster than previous fastest automated reaction mapping algorithms.

Keywords: Applied Algorithms, Automated Reaction Mapping, Cheminformatics.

1 Introduction

Computational simulations of chemistry are used by the chemical engineering community to solve a variety of problems and give insight to many problems, such as the analysis of combustion reactions. Automated reaction mapping is an important tool in cheminformatics where it may be used to classify reactions or validate large suites of reactions, called mechanisms. Improvements in computing power have made it possible to produce reaction mechanisms that contain hundreds of species and thousands of reactions. The size of reaction mechanisms is expected to continue to grow in order to provide more details about the chemistry they are modeling since are used in technical applications which require accurate and reliable simulations. Mechanism generation algorithms create all theoretically likely reactions which results in very large and unorganized mechanisms which must be reduced [1,2,3,4,5]. The mechanism reduction algorithms are computationally expensive and may take days to complete [6,7,8,9,10,11,12]. Prior to running a mechanism reduction algorithm a kineticist should sort the reactions, based on each reaction's classification, to verify that all of the important

* Research of the authors was funded in part by the National Science Foundation under Grant No. CNS–0931748. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 157–168, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Fig. 1. A simple chemical reaction: $H_2O_2 \Leftrightarrow OH + OH$

reactions and reaction classes are included. The kineticist may be required to run the mechanism generation algorithm and check the output multiple times prior to reducing the mechanism. *Since the kineticist may have to run the automated reaction mapping algorithms multiple times when generating a new mechanism, it is essential that these algorithms are efficient.*

A reaction may be represented as a collection of reactant and product graphs where a set of reactant graphs is transformed into a set of product graphs. *The reaction-mapping problem may be formulated as that of finding a mapping from the atoms of the reactant graphs to the atoms of the product graphs that minimizes the number of bonds broken or formed* [13]. For example, consider the reaction $H_2O_2 \Leftrightarrow OH + OH$ shown in Fig. 1. The optimal mapping will break the OO bond in the H_2O_2 reactant in order to form the two OH products. The general automated reaction mapping problem is known to be NP-Complete [13,14].

In this paper we present three algorithms that improve the runtime of the Constructive Count Vector (CCV) algorithm presented by [13,15], while maintaining solution optimality. The authors of [13,15] have proven their algorithms result in an optimal solution and our improvements do not affect optimality. The improvements proposed are based on (a) the use of a fast (but not 100% accurate) canonical labeling algorithm, (b) name reuse (i.e., storing intermediate results rather than recomputing), and (c) an incremental approach to canonical name computation. The three algorithms are Two Stage Constructive Count Vector (2-CCV), Two Stage Constructive Count Vector with Name Reuse (2-CCV NR), and Two Stage Constructive Count Vector with Name Reuse and Fast Degree Neighborhood Naming (2-CCV NR FDN). The improved algorithms find the optimal solution over ten times faster than CCV.

Section 2 presents background information and discusses related work. Section 3 presents our three algorithms for automated reaction mapping. Section 4 presents the experimental results obtained by testing our three algorithms on a variety of reaction mechanisms. Section 5 concludes the paper.

2 Background

2.1 Graph Isomorphism

A key step in automated reaction mapping is determining if two graphs are isomorphic. Two graphs, G_1 and G_2, are isomorphic if there is a bijection of vertices of G_1 and the vertices of G_2, $f : V(G_1) \rightarrow V(G_2)$, such that two vertices, u and v, are adjacent in G_1 if and only if $f(u)$ is adjacent to $f(v)$ in G_2. No efficient algorithm has been found to determine if two general graphs are isomorphic [16]. The problem of finding a canonical name for a graph is closely

related to the graph isomorphism problem. A canonical name of a graph is a unique label given to all isomorphic graphs. If a canonical name can be found for two graphs, the graphs can easily be checked for isomorphism by comparing the canonical names [17].

Chemical Graph Isomorphism and Canonical Labeling. Algorithms have been designed specifically for solving the chemical graph isomorphism problem. These chemical graph isomorphism algorithms have an exponential worst-case time complexity, but in practice are much faster.

One of the first canonical naming algorithms for chemical graphs was proposed by H.L. Morgan and is based on node connectivity and the creation of unambiguous strings which describe a molecule [18]. One of the most well known and fastest algorithms for determining chemical graph isomorphism is Nauty [19] which is based on finding the automorphism groups of a graph [20]. Another well known canonical naming algorithm for chemical graph isomorphism is *Signature* [21]. The algorithm finds a canonical name using extended valence sequences. Extended valence sequences are defined as a canonical representation of the topological environment of the considered atom up to a predefined height.

Random Graph Isomorphism and Canonical Labeling. Fast isomorphism testing algorithms have been developed for random graphs. These algorithms typically run in polynomial time, but have some probability of failure. In [22], the authors present a simple canonical labeling algorithm for random graphs based on vertex degree distributions. In [23], the authors present a linear time algorithm ($O(V+E)$, where V is the number of vertices and E is the number of edges) for the canonical labeling of a graph which is invariant under isomorphism.

2.2 Previous Automated Reaction Mapping Algorithms

Akutsu's Algorithm. In [14], the author provides two main algorithms for solving the automated reaction mapping problem by limiting the problem to a specific form of reactions: $XA + YB \Leftrightarrow XB + YA$. The first algorithm limits compounds to trees and has a worst-case time complexity of $O\left(n^{1.5}\right)$. The second algorithm does not limit the compounds to trees and has a worst-case time complexity of $O\left(n^{3}\right)$. Note that n is the maximum number of vertices of the compounds in the reaction.

Felix and Valiene's Algorithm. In [24], the authors reduce the automated reaction problem to a series of chemical substructure searches between the reactant and product graphs. This approach limits their automated reaction mapping algorithms to specific reaction classes. The authors identify four main classes of reactions which include combination reactions ($A + B \Leftrightarrow AB$), decomposition reactions ($AB \Leftrightarrow A + B$), displacement reactions ($A + BC \Leftrightarrow AC + B$), and exchange reactions ($AB + CD \Leftrightarrow AD + CB$).

Subgraph Isomorphism Based Algorithms. In [25,26], algorithms for automated reaction mapping are presented which are based on maximum common subgraph. The maximum common subgraph heuristic approaches are not

ideal since these solutions have no guarantee of finding the correct mapping [25] and the maximum common subgraph problem is known to be NP-hard [27]. It may also be difficult to find mapping rules consistent with multiple reaction formulas [14].

Maximum Common Edge Subgraph Based Algorithms. In [28,29], the authors extend a branch and bound algorithm called RASCAL. The algorithms are based on the maximum common edge subgraph problem. In the maximum common edge subgraph problem, the edges are mapped as opposed to vertices in the maximum common subgraph based approach. Since the maximum common edge subgraph problem is similar to the maximum common subgraph problem, it will have the same problems as any maximum common subgraph based approach.

Crabtree's Algorithms. In [13,15], the authors present five algorithms for automated reaction mapping which work for any valid chemical reaction. The first algorithm is a fast greedy heuristic which is not guaranteed to find an optimal solution. The second algorithm is an exponential-time exhaustive algorithm which is guaranteed to find the optimal solution. The remaining three algorithms use the chemical information of the reaction to intelligently generate bit patterns which represent bonds to be cut on the reactant and product graphs. These three algorithms produce an optimal solution.

The fastest algorithm presented in [13,15] which produces an optimal solution is the Constructive Count Vector (CCV) algorithm. The algorithm is based on a theorem which states that an identity chemical reaction has the same number of bonds with each bond symbol on each side of the equation. Note that in an identity chemical reaction the set of reactant molecules is isomorphic to the set of product molecules. A bond symbol refers to the atoms connected by each bond. For example a bond which connects a carbon and a hydrogen atom would have a CH bond symbol. The algorithm uses a count vector to determine the number of bonds, by symbol, on each side of the equation which should be cut. The algorithm then uses the count vector to create a bit pattern for each candidate equation. A bit pattern has one bit for each bond in the original equation where a bit set to '0' indicates the bond should remain and a bit set to '1' indicates the bond should be broken. After a candidate equation is created Nauty is used to look for isomorphic reactant and product species and therefore determine if the equation is mapped.

As an example, we will use reaction 318 from GRI-Mech [30]:

$$CH_3 + C_2H_4 \Leftrightarrow C_3H_7 \quad (1)$$

The reaction is shown in Fig. 2. Note that the label on each bond in the figure indicates its index in the generated CCV bit pattern.

The count vector for the reactants is given by $(CC, CH) = (1, 7)$ and the count vector for the products is given by $(CC, CH) = (2, 7)$. Therefore a count vector for the reactants and for the products which will produce a balanced bond equation while breaking the minimum number of bonds is given by $(CC, CH) = (0, 0)$ and $(CC, CH) = (1, 0)$, respectively. There are two product CC bonds (at

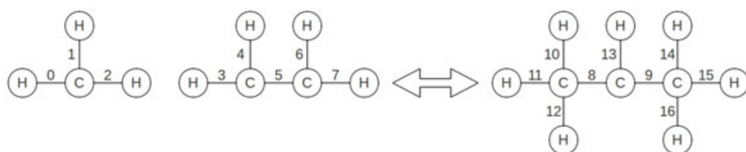


Fig. 2. GRI-Mech Reaction 318

indices 8 and 9) so there are two bit patterns which must be tested. Since neither bit pattern produces an identity chemical reaction, the CCV algorithm adds a single bond to the count vector on the reactant and product sides. The resulting reactant count vectors are (1,0) and (0,1). The resulting product count vectors are (2,0) and (1,1). These count vectors result in 99 bit patterns which must be tested and 24 of those bit patterns result in an identity chemical reaction. The CCV algorithm is able to stop after finding the first mapping or find all mappings of minimum cost.

The worst-case complexity of CCV is $O(F(l, r, b)CN(n))$, where $CN(n)$ represents the time complexity of the canonical naming algorithm used, n represents the number of atoms in the reaction, l represents the bond symbol vector for the reactants, r represents the bond symbol vector for the products, and $F(l, r, b)$ represents the number of bit patterns with b bonds that result in balanced bond symbols.

The CCV algorithm is novel because it is guaranteed to find the optimal solution and it does not have the limitations that previous algorithms have placed on the problem. Other reaction mapping algorithms are not guaranteed to find the best mapping or they will not work for all classes of reactions. Since the CCV algorithm is the fastest algorithm presented in [13,15] and CCV provides an optimal solution for a general, well-defined optimization problem we will be using CCV as our point of comparison.

3 Improved Automated Reaction Mapping

3.1 Fast Canonical Labeling Using Degree Neighborhoods

In the following algorithms we will name molecules using a fast chemical graph canonical naming algorithm that is not 100% accurate. The algorithm presented is similar to the random graph canonical naming algorithms presented in [22,23].

The main idea of the Degree Neighborhood (DN) algorithm is to assign each atom a name based on its symbol and degree and the symbol and degree of each of its neighbors. The names of each atom are then used to assign the name to the molecule. The DN algorithm is able to assign one name to a collection of molecules at the same time which allows us to give a single canonical name to all of the reactant or all of the product molecules.

For example, consider the CH_3O molecule in Fig. 3(a). We label each atom, using its symbol and degree, Fig. 3(b). We then add to each atom's name, the symbol and degree of its neighbors lexicographically, Fig. 3(c). Now that each

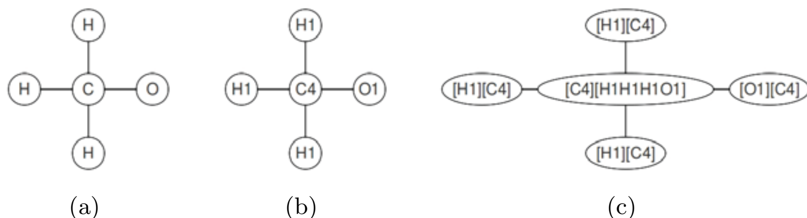


Fig. 3. Degree Neighborhood Canonical Labeling

atom is named, we lexicographically sort the atom names to create the name for the molecule. The resulting molecule name is therefore $[[C4][H1H1H1O1]][[H1][C4]][[H1][C4]][[H1][C4]][[O1][C4]]$.

Each atom is named according to its own symbol and degree in addition to its lexicographically sorted degree neighborhood which guarantees two isomorphic molecules have the same canonical name. There is no guarantee two non-isomorphic molecules do not have the same name. Experimentally we have found that DN correctly distinguishes between isomorphic and non-isomorphic molecules over 99% of the time.

3.2 Two Stage Constructive Count Vector

CCV only generates bit patterns that correspond to potential mappings which have balanced bond symbols. For each bit pattern that is generated, the algorithm creates a new equation, names the reactant and product molecules using Nauty and then checks if it is an identity chemical reaction. Our first optimization, named Two Stage Constructive Count Vector (2-CCV), prevents using Nauty unless we suspect resulting equation is an identity chemical reaction. The 2-CCV algorithm names each new equation in 2 stages. The first stage naming is done by the DN algorithm and the second stage naming is done by Nauty, which is only invoked if DN returns a match.

As an example, we will use the same reaction as before (Fig. 2). The initial name of the reactant molecules is:
$[[C3][C3H1H1]][[C3][C3H1H1]][[C3][H1H1H1]][[H1][C3]][[H1][C3]]$
$[[H1][C3]][[H1][C3]][[H1][C3]][[H1][C3]][[H1][C3]]$.

The initial name of the product molecules is:
$[[C3][C4C4H1]][[C4][C3H1H1H1]][[C4][C3H1H1H1]][[H1][C3]]$
$[[H1][C4]][[H1][C4]][[H1][C4]][[H1][C4]][[H1][C4]][[H1][C4]]$.

Consider the case from the previous example (Fig. 2) where the bit pattern breaks the bond labeled 8. The name computed for the reactants will not change and the new name for the product molecules is:
$[[C2][C4H1]][[C3][H1H1H1]][[C4][C3H1H1H1]][[H1][C2]][[H1][C3]]$
$[[H1][C3]][[H1][C3]][[H1][C4]][[H1][C4]][[H1][C4]]$.

Clearly the reactant and product names are not the same so we do not need to check this bit pattern using Nauty.

The 2-CCV approach reduces the number of times the algorithm utilizes Nauty, the more expensive chemical graph canonical labeling algorithm. It is also easier to check a potential mapping during the first stage mapping since all of the reactant or product molecules are named together. The first stage mapping can be checked using a single string comparison rather than matching each reactant molecule with a product molecule.

The asymptotic worst-case complexity of 2-CCV is

$$
\begin{aligned}
&O((F(l,r,b)-(m+p))DN(n)+(m+p)(DN(n)+CN(n))\\
=&O((F(l,r,b)-(m+p))DN(n)+(m+p)(CN(n))
\end{aligned}
$$

where $CN(n)$, $F(l,r,b)$, l, r, and b were previously defined and $DN(n)$ represents the time complexity of the degree neighborhood algorithm, m represents the number of optimal mappings and p represents the probability the degree neighborhood algorithm results in an error. Notice that the worst-case complexity of 2-CCV is worse than the worst-case complexity of CCV since it is possible to check each candidate using both DN and Nauty, but 2-CCV performs significantly better than CCV because p is very small in practice. Note that the worst-case complexity of the degree neighborhood algorithm is $O(n \lg n)$.

3.3 Two Stage Constructive Count Vector with Name Reuse

A second optimization was added to store the canonical names which are computed for each candidate mapping. The main idea is that the same reactants and products get generated multiple times during the algorithm. Rather than recomputing the names, we store and reuse them. The Two Stage Constructive Count Vector with Name Reuse (2-CCV NR) creates a hash map for the reactants and hash map for the products for each stage to store the canonical names since many of the candidate mappings generated break the same reactant or product bonds. Although 2-CCV NR is faster in practice than 2-CCV, the worst-case complexity is the same for both algorithms.

As an example, we will use the same reaction as before (Fig. 2). During the second iteration we will test breaking the CC reactant bond 14 times for various combinations of breaking CC and CH product bonds. After storing the reactant's name when testing the first candidate we will look up the reactant's name for the remaining 13 candidates.

3.4 Two Stage Constructive Count Vector with Name Reuse and Fast Degree Neighborhood Naming

The final optimization, Two Stage Constructive Count Vector with Name Reuse and Fast Degree Neighborhood Naming (2-CCV NR FDN), generates the DN name for a new candidate by updating the DN name from a previously computed candidate rather than computing it from scratch. When a bond is broken it affects the names of the atoms connected by the bond and their neighbors. Note that chemical graphs have bounded valence so the number of neigh-

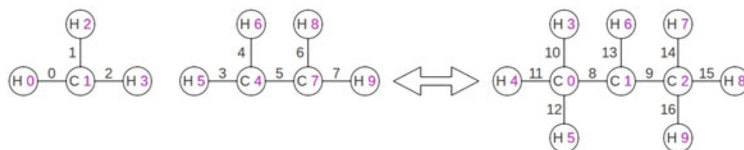


Fig. 4. GRI-Mech Reaction 318 with Node Indices

bors each atom has is limited. The names of the remaining atoms remain unchanged. We generate the name for a new candidate bit pattern from a previously computed bit pattern which broke one less bond. Note that each set of count vectors breaks one more reactant and one more product bond than the previous set of count vectors. For example, in reaction 318 from GRI-Mech [30] the first set of count vectors break 0 reactant bonds and 1 product bond. The second set of count vectors break 1 reactant bond and 2 product bonds.

In order to use DN name updates, we store an array of atom names in the stage 1 hash maps rather than the DN name for all of the molecules. Each atom of the reactants and products has an index which quickly matches an atom to its stored name. For example, the indices for reaction 318 from GRI-Mech are shown in Fig. 4. Once all of the atom names have been updated, they can be lexicographically sorted to produce the canonical name for all of the molecules.

The first step in updating the DN name comes from updating the names of the atoms connected by the broken bond. The affected atoms must have their degree reduced by one since they now have one less neighbor. In addition the affected atoms must be removed from each other's neighbor's list. Note that both of these changes are completed using string manipulations.

The second step in updating the DN names comes from updating the names of the atoms which are neighbors to the atoms connected by the broken bond. Without loss of generality, assume the broken bond affected atoms a_i and a_j, where $i \neq j$. We look at each bond that is connected to atom a_i. If a bond (e.g., the bond connects a_i and a_k) has not been previously broken (i.e., its bit in the candidate bit pattern is set to '0') then the algorithm updates the neighbor list for atom a_k. The neighbor list is updated by replacing the first occurrence of a_i's old symbol and degree with a_i's new symbol and degree. The process is repeated for atom a_j. Note that these node name changes are completed using string manipulations.

As an example we will use the product molecule from GRI-Mech Reaction 318 (Fig. 4). We start with the array of node canonical labels for the molecule, given by the array: 0: $[C4][C3H1H1H1]$, 1: $[C3][C4C4H1]$, 2: $[C4][C3H1H1H1]$, 3: $[H1][C4]$, 4: $[H1][C4]$, 5: $[H1][C4]$, 6: $[H1][C3]$, 7: $[H1][C4]$, 8: $[H1][C4]$, 9: $[H1][C4]$.

Suppose we want to break the bond with index 8 which connects the atoms with index 0 and index 1. The algorithm will retrieve each node's canonical label and update it. The atom at index 0 was previously named $[C4][C3H1H1H1]$. The portion of the label which contains its symbol and degree will change from $C4$ to $C3$ since the atom now has one less neighbor. The portion of the label

which contains the neighbor list must remove the reference to the atom at index 1. Therefore the substring $C3$ will be removed from the neighbor list. The new canonical name for the atom at index 0 is $[C3][H1H1H1]$. Similarly the new canonical name for the atom at index 1 is $[C2][C4H1]$. Now we must update the neighbors of the atom at index 1. The first neighbor is the atom at index 3 which has the name of $[H1][C4]$. The algorithm replaces $C4$ in the neighbor list with $C3$ resulting in the new name of $[H1][C3]$. Similarly the atoms at index 4 and index 5 are renamed $[H1][C3]$ and the atom at index 6 is renamed $[H1][C2]$.

Although the worst-case complexity of 2-CCV NR FDN remains the same as 2-CCV NR, in practice 2-CCV NR FDN performs much faster. The 2-CCV NR FDN algorithm reduces the time it takes to compute the DN canonical name of the reactants and products during stage 1. In addition, 2-CCV NR FDN does not have to generate the candidate equation unless, from stage 1, we suspect the equation is mapped.

4 Experimental Results

The experiments were carried out on a computer running Windows Vista Home Premium with a 2.66 GHz Intel Core 2 Quad Processor and 4 GB of RAM. The code was written in Java and developed with JDK 1.4. The time statistics provided are for relative comparison purposes only since Java uses automatic garbage collection that is not controlled by the programmer. Note that for all of the databases we used the Nauty [19] chemical graph canonical naming algorithm to test for isomorphism. Although we tested the code on a variety of mechanisms only a few are included due to space considerations. The results are summarized in Table 1, where the column labeled 'single' refers to finding a single mapping of minimum cost and the column labeled 'all' refers to finding all mappings of minimum cost.

Note that CCV is guaranteed to find an optimal solution when mapping a reaction. For each of the databases tested, we verified our algorithms produced the same output as CCV.

The Colorado School of Mines (CSM) oxidation and pyrolysis mechanisms were derived from published oxidation and pyrolysis mechanisms [31,32]. The CSM oxidation mechanism contains 3544 reactions and the CSM pyrolysis mechanism contains 1707 reactions. Notice that the 2-CCV NR FDN algorithm is over

Table 1. Runtime Results

	CSM Oxidation		CSM Pyrolysis		LLNL	
	Single (sec)	All (sec)	Single (sec)	All (sec)	Single (sec)	All (sec)
CCV	305.6	1078.7	277.3	1053.7	16122.9	40616.3
2-CCV	177.8	627.1	162.8	588.8	9758.7	23730.4
2-CCV NR	118.1	404.1	108.7	378.9	6125.6	15401.5
2-CCV NR FDN	22.1	65.9	15.4	57.8	1160.5	3215.1

18 times faster than the CCV algorithm for the CSM pyrolysis mechanism regardless of whether we are finding a single mapping or all mappings of minimum cost. The 2-CCV NR FDN algorithm is over 13 times faster than the CCV algorithm for the CSM oxidation mechanism when finding a single mapping of minimum cost and over 16 times faster than the CCV algorithm when finding all mappings of minimum cost.

The database provided by Lawrence Livermore National Laboratory (LLNL) [33] models combustion and ignition phenomena for normal heptane. The connectivity data for the molecules was added by students in the Chemical Engineering department at Colorado School of Mines for an early version of the database that contained errors [13]. Using the provided information we were able to map over 4000 reactions from the LLNL database. Notice that the 2-CCV NR FDN algorithm runs over 12 times faster than CCV regardless of whether we finding a single mapping or all mappings of minimum cost.

5 Conclusion

In conclusion, this paper presented three algorithms to solve the automated reaction mapping problem that are based on (a) the use of a fast (but not 100% accurate) canonical labeling algorithm, (b) name reuse (i.e., storing intermediate results rather than recomputing), and (c) an incremental approach to canonical name computation. The algorithms presented in this paper are significantly faster in practice than previous reaction mapping algorithms. *The time to map the reactions from the LLNL database [33] previously took over 11 hours using CCV, but using 2-CCV NR FDN it now takes less than 1 hour to complete.* Improved automated reaction mapping algorithms are essential for the growing needs of the cheminformatics and bioinformatics community.

References

1. Muharam, Y., Warnatz, J.: Kinetic Modelling of the Oxidation of Large Aliphatic Hydrocarbons Using an Automatic Mechanism Generation. Phys. Chem. Chem. Phys. 9, 4218–4229 (2007)
2. Matheu, D., Grenda, J.: A Systematically Generated, Pressure-Dependent Mechanism for High-Conversion Ethane Pyrolysis. 1. Pathways to the Minor Products. J. Phys. Chem. 109, 5332–5342 (2005)
3. Cartensen, H., Dean, A.M.: Rate Constant Rules for the Automated Generation of Gas-Phase Reaction Mechanisms. J. Phys. Chem. 113, 367–380 (2009)
4. Nemeth, A., Vidoczy, T., Heberger, K., Kuti, Z., Wagner, J.: MECHGEN: Computer Aided Generation and Reduction of Reaction Mechanisms. J. Chem. Inf. Comput. Sci. 42, 208–214 (2002)
5. Buda, F., Bounaceur, R., Warth, V., Glaude, P.A., Fournet, R., Battin-Leclerc, F.: Progress Toward a Unified Detailed Kinetic Model for the Autoignition of Alkanes from C4 to C10 Between 600 and 1200 K. Combust. Flame. 142, 170–186 (2005)
6. Straube, R., Flockerzi, D., Muller, S.C., Hauser, J.B.: Reduction of Chemical Reaction Networks Using Quasi-Integrals. J. Phys. Chem. 109, 441–450 (2005)

7. Pepiot-Desjardins, P., Pitsch, H.: An Efficient Error-propagation-based Reduction Method for Large Chemical Kinetic Mechanisms. Combust. Flame. 154, 67–81 (2008)
8. Liang, L., Stevens, J., Raman, S., Farrell, J.: The Use of Dynamic Adaptive Chemistry in Combustion Simulation of Gasoline Surrogate Fuels. Combust. Flame. 156, 1493–1502 (2009)
9. Nagy, T., Turanyi, T.: Reduction of Very Large Reaction Mechanisms Using Methods Based on Simulation Error Minimization. Combust. Flame. 156, 417–428 (2009)
10. Sun, W., Chen, Z., Gou, X., Yiguang, J.: A Path Flux Analysis Method for the Reduction of Detailed Chemical Kinetic Mechanisms. Combust. Flame. 157, 1298–1307 (2010)
11. Shi, Y., Ge, H., Brakora, J., Reitz, R.: Automatic Chemistry Mechanism Reduction of Hydrocarbon Fuels for HCCI Engines Based on DRGEP and PCA Methods with Error Control. Energy & Fuels 24, 1646–1654 (2010)
12. Kovacs, T., Zsely, I., Kramarics, A., Turanyi, T.: Kinetic Analysis of Mechanisms of Complex Pyrolytic Reactions. J. Anal. Appl. Pyrolysis. 79, 252–258 (2007)
13. Crabtree, J.D., Mehta, D.P.: Automated Reaction Mapping. J. Exp. Algorithmics. 13, 1.15–1.29 (2009)
14. Akutsu, T.: Efficient Extraction of Mapping Rules of Atoms from Enzymatic Reaction Data. J. Comput. Biol. 11, 449–462 (2004)
15. Crabtree, J., Mehta, D., Kouri, T.: An Open-Source Java Platform for Automated Reaction Mapping. J. Chem. Inf. Model. 50(9), 1751–1756 (2010)
16. Pemmaraju, S., Skiena, S.: Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Cambridge University Press, New York (2003)
17. Babai, L., Luks, E.: Canonical labeling of graphs. In: STOC 1983: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, pp. 171–183. ACM, New York (1983)
18. Morgan, H.L.: The Generation of a Unique Machine Description for Chemical Structures - A Technique Developed at Chemical Abstracts Service. J. Chem. Doc. 5(2), 107–113 (1965)
19. B. McKay. No automorphisms, yes? (2004), http://cs.anu.edu.au/~bdm/nauty/
20. McKay, B.: Practical Graph Isomorphism. Congr. Numer. 30, 45–87 (1981)
21. Faulon, J.-L., Collins, M.J., Carr, R.D.: The Signature Molecular Descriptor. 4. Canonizing Molecules Using Extended Valence Sequences. J. Chem. Inf. Model. 44(2), 427–436 (2004)
22. Babai, L., Erdos, P., Selkow, S.: Random Graph Isomorphism. Siam J. Comput. 9(3), 628–635 (1980)
23. Czajka, T., Panduranga, G.: Improved Random Graph Isomorphism. Journal of Discrete Algorithms 6, 85–92 (2008)
24. Felix, L., Valiente, G.: Efficient Validation of Metabolic Pathway Databases. In: Proc. 6th Int. Symp. Computational Biology and Genome Informatics, pp. 1209–1212 (2005)
25. Arita, M.: Metabolic Reconstruction Using Shortest Paths. Simulation Practice and Theory 8(2), 109–125 (2000)
26. Hattori, M., Okuno, Y., Goto, S., Kanehisa, M.: Development of a Chemical Structure Comparison Method for Integrated Analysis of Chemical and Genomic Information in the Metabolic Pathways. J. Am. Chem. Soc. 125(1), 11853–11865 (2003)
27. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W.H. Freeman & Co., New York (1990)

28. Korner, R., Apostolakis, J.: Automatic Determination of Reaction Mappings and Reaction Center Information. 1. The Imaginary Transition State Energy Approach. Journal of Chemical Information and Modeling 48(6), 1181–1189 (2008)
29. Apostolakis, J., Sacher, O., Korner, R., Gasteiger, J.: Automatic Determination of Reaction Mappings and Reaction Center Information. 2. Validation on a Biochemical Reaction Database. Journal of Chemical Information and Modeling 48(6), 1190–1198 (2008)
30. Gas Research Institute. Gri-mech 3.0, http://www.me.berkeley.edu/gri-mech/
31. Naik, C.V., Dean, A.M.: Detailed Kinetic Modeling of Ethane Oxidation. Combust. Flame. 145, 16–37 (2006)
32. Randolf, K.L., Dean, A.M.: Hydrocarbon Fuel Effects in Solid-oxide Fuel Cell Operation: An Experimental and Modeling Study of n-hexane Pyrolysis. Phys. Chem. Chem. Phys. 9, 4245–4258 (2007)
33. Curran, H.J., Gaffuri, P., Pitz, W.J., Westbrook, C.K.: A comprehensive modeling study of n-heptane oxidation. Combust. Flame. 114(1-2), 149–177 (1998)

An Experimental Evaluation of Incremental and Hierarchical k-Median Algorithms*

Chandrashekhar Nagarajan[1] and David P. Williamson[2]

[1] Yahoo! Inc., Sunnyvale, CA, 94089, USA
cn54@yahoo-inc.com
[2] Cornell University, Ithaca, NY, 14853, USA
dpw@cs.cornell.edu

Abstract. In this paper, we consider different incremental and hierarchical k-median algorithms with provable performance guarantees and compare their running times and quality of output solutions on different benchmark k-median datasets. We determine that the quality of solutions output by these algorithms for all the datasets is much better than their performance guarantees suggest. Since some of the incremental k-median algorithms require approximate solutions for the k-median problem, we also compare some of the existing k-median algorithms' running times and quality of solutions obtained on these datasets.

1 Introduction

A company is building facilities in order to supply its customers. Because of limited capital, it can only build a few at this time, but intends to expand in the future in order to improve its customer service. Its plan for expansion is a sequence of facilities that it will build in order as it has funds. Can it plan its future expansion in such a way that if it opens the first k facilities in its sequence, this solution is close in value to that of an optimal solution that opens any choice of k facilities? The company's problem is the *incremental* k-median problem, and was originally proposed by Mettu and Plaxton [10].

The standard k-median problem has been the object of intense study in the algorithms community in recent years. Given the locations of a set of facilities and a set of clients in a metric space, and a parameter k, the *k-median problem* asks to find a set of k facilities to *open* such that the sum of the distances of the clients to the nearest open facility is minimized. Since the metric k-median problem is NP-hard [8], many researchers have focused on obtaining approximation algorithms for it. An α-approximation algorithm for a minimization problem runs in polynomial time and outputs a solution whose cost is at most α times the cost of the optimal solution. The factor α is sometimes called the *approximation factor* or *performance guarantee* of the algorithm. A solution for which the cost is at most α times the optimal cost is sometimes called *α-approximate.* The best approximation algorithm known for this problem has a performance

* Supported in part by NSF grant CCF 0514528.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 169–180, 2011.
© Springer-Verlag Berlin Heidelberg 2011

guarantee of $3 + \epsilon$ and is due to Arya, Garg, Khandekar, Meyerson, Munagala and Pandit [2]; it is based on a local search heuristic.

In the incremental k-median problem, we are given the input of the k-median problem without the parameter k and must produce a sequence of the facilities. For each k, consider the ratio of the cost of opening the first k facilities in the ordering to the cost of an optimal k-median solution. The goal of the problem is to find an ordering that minimizes the maximum of this ratio over all values of k. An algorithm for the problem is said to be *α-competitive* if the maximum of the ratio over all k is no more than α. This value α is called the *competitive ratio* of the algorithm. Mettu and Plaxton [10] gave a 29.86-competitive algorithm for the incremental k-median problem. Later Lin, Nagarajan, Rajaraman and Williamson [9] gave deterministic 16-competitive and randomized 10.88-competitive algorithms for the incremental k-median problem[1]. Their algorithms use either a k-median approximation algorithm or a Lagrangean Multiplier Preserving (LMP) facility location algorithm as a black box.

We also consider algorithms for the hierarchical k-median problem. In hierarchical clustering, we give clusterings with k clusters for all values of k by starting with each point in its own cluster and repeatedly merging selected pairs of clusters until all points are in a single cluster. We also consider a variation of this problem in which each cluster has a point designated as its center, and when we merge two clusters together to form a single cluster, one of the two centers becomes the center of the new cluster. Given some objective function on a k-clustering, again we would like to ensure that for any k, the cost of our k-clustering obtained in this way is not too far away from the cost of an optimal k-clustering. For the hierarchical k-median problem, the objective function for the k-clustering is its k-median cost; that is, the sum of the distances of each point to its cluster center. Plaxton [11] gave a 238.88-competitive algorithm for the problem. Lin et al. [9] later gave deterministic 40.42-competitive and randomized 20.06-competitive algorithms for the problem. Their algorithms again use either a k-median approximation algorithm or a LMP facility location algorithm as a black box.

In this paper, we consider the performance of these incremental and hierarchical k-median algorithms on different k-median benchmark datasets and compare their running times and quality of output solutions. Since the algorithms of Lin et al. require a k-median approximation algorithm or a LMP facility location algorithm as a black box, we also compare the performance of some of the existing k-median and LMP facility location algorithms. In particular, we implement five different k-median and LMP facility location algorithms. The first one is the single swap local search algorithm by Arya et al. [2], which gives 5-approximate solutions. We also consider the linear program (LP) rounding algorithm of Charikar, Guha, Tardos and Shmoys [4] which rounds the LP optimum to get 8-approximate solutions. Jain, Mahdian, Markakis, Saberi and Vazirani [7] give a greedy dual-fitting Lagrangean Multiplier Preserving (LMP)

[1] Some of the results of Lin et al. were obtained independently by Chrobak, Kenyon, Noga, and Young [5].

Facility Location (FL) algorithm which gives 2-approximate k-median solutions for some values of k. We also consider the standard k-median linear program and solve it optimally using CPLEX. The optimal solution can be fractional but still gives a good lower bound for the k-median problem. We also solve the k-median integer program optimally using CPLEX even though the algorithm is not polynomial time. These linear and integer programs give us bounds on the quality of the solutions of the other algorithms.

Given these algorithms, we implement several variants of the Lin et al. algorithms for the incremental k-median problem. We implement their algorithm using the Arya et al. local search algorithm for k-median, the Charikar et al. LP rounding algorithm for k-median, and the Jain et al. greedy algorithm which is an LMP algorithm for facility location. Additionally, we implement the original algorithm of Mettu and Plaxton for the incremental k-median problem. We are able to use the linear and integer programming solutions to bound the quality of the results we obtain.

We also implement several variants of the Lin et al. algorithms for hierarchical k-median problem. Again, we implement their algorithm using the Arya et al. local search algorithm for k-median, the Charikar et al. LP rounding algorithm, and the Jain et al. greedy algorithm. Additionally, we implement Plaxton's algorithm for the hierarchical k-median problem. Plaxton's algorithm requires an incremental k-median algorithm as a black box, and originally used the algorithm of Mettu and Plaxton as a subroutine. We implement this variant of Plaxton's algorithm, and also a variant that uses Lin et al.'s algorithm given the Arya et al. local search algorithm.

We test our algorithms on 43 different k-median instances drawn from the literature. In particular, we use forty instances from the OR Library [3], two instances from Galvão and ReVelle [6], and one instance from Alp, Erkut, and Drezner [1].

From the results we obtained we determine that all these algorithms perform much better in terms of quality of solution than their respective competitive/approximation ratios suggest. In particular, while we know of no polynomial-time algorithm with a competitive ratio better than 10 for the incremental and hierarchical median k-median problems, we typically obtained results which were within 10% of the k-median LP relaxation for incremental problems and 20% of the k-median LP relaxation for hierarchical k-median problems. We find this quite surprising in view of the strong constraints required on the structure of solutions for the incremental and hierarchical problems.

The algorithms of Mettu and Plaxton for incremental k-median and Plaxton for hierarchical k-median produce solutions that are not as good as those of Lin et al.; however, our implementation of the Mettu-Plaxton algorithm is significantly faster than our implementations of the Lin et al. algorithms, at least in part because the Lin et al. algorithms require approximate solutions of the k-median problem for all values of k.

Our paper is structured as follows. In Section 2, we sketch various algorithms we implemented. In Section 3, we discuss the datasets we used. In Section 4, we

give the experimental results we obtained. In Section 5, we give our conclusions as well as some open problems prompted by our work. For space reasons, detailed statements of the algorithms and complete tables of results are omitted, and will appear in the full version of the paper.[2]

2 Algorithms

In this section, we discuss the various algorithms we implemented for the k-median, incremental k-median, and hierarchical k-median problems respectively.

2.1 The k-Median Problem

In this section we consider five different algorithms for the k-median problem: the single swap local search algorithm by Arya et al. [2]; the linear program (LP) rounding algorithm of Charikar et al. [4] which rounds the LP optimum to get an integer solution which is no more than 8 times the cost of the optimal LP solution; the Jain et al. [7] greedy dual-fitting Lagrangean Multiplier Preserving (LMP) Facility Location (FL) algorithm, which gives 2-approximate k-median solutions for some values of k; the standard k-median linear program, which we solve optimally using CPLEX; and the k-median integer program, which we also solve optimally using CPLEX even though the algorithm is not polynomial time. The optimal solution to the linear program can be fractional but still gives a good lower bound for the k-median problem. We now discuss each of these algorithms in turn. For space reasons, we cannot give full descriptions.

Local Search Algorithm of Arya et al. We consider the Arya et al.'s ([2]) single swap local search algorithm which computes a 5-approximate solution. The local search algorithm proceeds by starting with an arbitrary solution and repeatedly doing *valid swaps* on the current solution till no more valid swaps exist. A swap closes a facility in the current solution and opens a facility that was previously closed. A swap is considered valid if the cost of the new solution after swapping is less than the cost of the solution before swapping.

Arya et al. proved that the local search algorithm can be made to run in time polynomial in the input size by considering a swap as valid only if it improves the cost of the solution by a certain factor. However, for simplicity, we consider any cost-improving swap as a valid swap. We run this local search algorithm for each cardinality k. After this procedure we have locally optimal solutions for each value of k.

We do not implement the multi-swap (swaps involving more than one facilities) local search algorithm by Arya et al. because of its high running time even though it gives better approximation guarantee of $3+\epsilon$. We use the locally optimal solution of cardinality $k-1$ as a starting solution for the local search

[2] A more complete abstract of the paper, including full explanations of the algorithms, and full tables of results and running times, can be found at `http://www.orie.cornell.edu/~dpw/incexp.pdf`

iteration for cardinality k. Since this solution is already a good solution for cardinality k we reduce the running times of the subsequent iterations. On average this improves the running times of local search by about 40%.

LP rounding algorithm of Charikar et al. We consider the LP rounding algorithm of Charikar et al. [4] which takes as input the fractional optimal solution of the standard LP relaxation $(k-P)$ of the k-median problem and produces an integer solution that is no more than 8 times the cost of LP optimum.

The algorithm is as follows. It starts with the optimal LP fractional solution for a particular value of k. First, the algorithm simplifies the problem instance by consolidating nearby clients and combining their demands such that the clients with nonzero demands are far from each other the resulting problem instance. It then simplifies the structure of the optimal fraction solution by consolidating nearby fractional facilities. The resulting solution has nonzero fractional value only on facilities with nonzero demands and the LP variables for the facilities are no less than $\frac{1}{2}$. The algorithm then modifies this solution to a solution where the LP variables for the facilities take values of only 0, $\frac{1}{2}$ and 1. It then opens no more than k of these facilities, selecting them based on their distance to other facilities with positive LP value.

Greedy LMP FL Algorithm of Jain et al. Jain et al. [7] give a LMP greedy dual-fitting algorithm for the facility location problem. In this algorithm, we maintain a dual value v_j for every client which is its total contribution to getting connected to a open facility. Some part of this dual v_j pays for the j's connection cost and the remainder is paid toward facility opening costs. We increase the duals of the clients uniformly and open a facility when a facility has enough contribution from the clients to match the facility opening cost. We say a client is connected to a facility if the connection cost is paid for by its dual value. We stop increasing the dual for a client if it is connected to a open facility.

Since this facility location algorithm is a LMP 2-approximation algorithm for the facility location (FL) problem, we can obtain something called a *bounded envelope* for the k-median problem as described in Lin et al. [9]. The bounded envelope gives 2-approximate solutions for the k-median problem for some values of k as well as a corresponding piecewise linear lower bound on the values of k-median solutions for all values of k, where the breakpoints of the lower bounds occur at values of k for which we have 2-approximate solutions. Lin et al. give a procedure for computing the bounded envelope given the LMP FL algorithm.

Solving Linear Program using CPLEX. We solve the linear programming relaxation $(k-P)$ of the standard k-median problem using the CPLEX solver.

To speed up the running time of the linear program solver, we tried to give the optimal solution of $(k-1)^{th}$ run as an initial starting solution to the iteration of cardinality k for all values of k. But there was no significant improvement of the running times of the linear programs on average.

Solving Integer Program using CPLEX. We solve the integer program $(k-IP)$ optimally using the CPLEX solver; $(k-IP)$ is the same as $(k-P)$ except that we require the decision variables to be 0-1. The CPLEX solver provides a way to give a good initial guess to the solver so that it can prune many low quality solutions. We give the optimal integer solution with $k-1$ facilities as an initial guess for the CPLEX integer program iteration with cardinality k. As the optimal solution for the k-median problem for a smaller value of cardinality is a feasible solution for the k-median problem with larger cardinality, the initial guess is feasible. Even though this makes the solver find the optimal integral solution faster in some cases, it does not work in all cases and on average the improvement in running time is not significant.

2.2 Incremental *k*-Median

In this section we briefly explain the Mettu and Plaxton's incremental k-median algorithm and Lin et al.'s incremental k-median algorithm.

Mettu and Plaxton's Algorithm. Mettu and Plaxton's [10] incremental k-median algorithm uses a hierarchical greedy approach to choose the next facility in the incremental order to be opened. The basic idea behind this approach is as follows. Rather than selecting the next point in the ordering based on a single greedy criterion, they greedily choose a region and then recursively choose smaller regions till they arrive at a single facility which then becomes the next facility to open. Thus the choice of the next facility is influenced by a sequence of greedy criteria addressing successive finer levels of granularity.

Lin et al.'s incremental *k*-median algorithm. We implement the incremental algorithm AltIncApprox of Lin et al. [9] for the incremental k-median problem on these datasets. We use Arya et al.'s local search algorithm with single swaps and the LP rounding technique of Charikar et al. to generate good k-median solutions for all possible k for each of these datasets. We bucket these solutions into buckets of geometrically increasing cost. We take the costliest solution from each bucket. We then consider each of these solutions in order of decreasing number of medians, and use each such solution to find another solution with the same number of medians that is contained with the next larger solution. This gives us a sequence of k-median solutions such that any smaller solution is a subset of any larger solution. This sequence of solutions gives a natural ordering of the facilities.

We also implement the incremental algorithm BoundedIncApprox of Lin et al. [9] using the k-median bounded envelope obtained by running the Jain et al. algorithm on the datasets. By using the 2-approximate solutions obtained from this algorithm for some values of k, we can apply the procedure given above to obtain an ordering of the facilities.

2.3 Hierarchical *k*-Median

We test the hierarchical k-median algorithms of Lin et al. [9] against the previously known hierarchical k-median algorithm by Plaxton [11].

Plaxton's Algorithm. Plaxton's algorithm takes in an incremental k-median solution as input and finds a *parent* function for each facility this incremental ordering. A hierarchical k-median solution obtained from an ordering can be considered as solutions obtained by repeatedly closing the last open facility in ordering and assigning its clients to an earlier facility. This mapping is exactly captured by the parent function in the Plaxton's algorithm. A parent function for an ordering maps every facility in the order to a facility that is earlier in the ordering. The parent of a facility is the facility that its clients will get assigned to when the facility is closed.

Plaxton's parent function is assigned as follows: Given an incremental k-median solution to the problem, a parent is assigned to every facility in the reverse order of the incremental solution. The parent of a facility f is determined by the earliest facility in the ordering that is either the closest facility or satisfies a certain equation. The equation essentially finds a facility whose distance to f is no more than the average distance of f's clients to f.

We run the Plaxton's parent function algorithm on the incremental k-median solutions given by running the Mettu and Plaxton's algorithm and AltIncApprox algorithm using Arya et al.'s local search solutions on the datasets.

Lin et al.'s hierarchical k-median algorithm. We run the generic algorithm AltIncApprox of Lin et al. [9] for the hierarchical k-median problem on the datasets using different k-median algorithms as black box. We use Arya et al.'s local search algorithm and Charikar et al.'s LP rounding algorithm to generate good k-median solutions. We also implement the incremental algorithm BoundedIncApprox of Lin et al. [9] using the k-median bounded envelope obtained by running Jain et al. algorithm on the datasets. As in the incremental k-median algorithm of Lin et al. we must find approximate solutions to the k-median problem, which we then put in buckets of geometrically increasing cost, then take the costliest solution from each bucket. We consider these solutions in order of decreasing size, and use each solution to find a k-clustering that is consistent with a hierarchical clustering on the larger solutions already considered.

3 Datasets

In our experiments we use these following datasets for the comparison of k-median, incremental k-median and the hierarchical k-median algorithms.

1. *OR Library:* These 40 datasets of the uncapacitated k-median problems are part of the OR Library [3], which is a collection of test datasets for a variety of OR problems created by J. E. Beasley. These 40 test problems are named *pmed*1, *pmed*2, ..., *pmed*40 and their sizes range from $n = 100$ to 900. As noted in [3], we apply Floyd's algorithm on the adjacency cost matrix in order to obtain the complete cost matrix.

2. *Galvão:* This set of instances (*Galvão100* and *Galvão150*) is obtained from the work of Galvão and ReVelle [6]. Even though the sizes of these datasets are small ($n = 100$ and $n = 150$), the integrality gaps for some values of k (number of medians) are larger than traditional datasets.
3. *Alberta:* This dataset is generated from a 316-node network using all population centers in Alberta (see Alp, Erkut and Drezner [1]) where the distances are computed using the shortest path metric on the actual road network of Alberta.

4 Experimental Results

4.1 The *k*-Median Problem

In this section we compare the performance in terms of running times and quality of solutions of five different algorithms on the datasets described: CPLEX solver for the k-median linear program, CPLEX solver for k-median integer program, Arya et al.'s single swap local search algorithm, Charikar et al.'s LP rounding algorithm and the bounded envelope of Jain et al.'s greedy algorithm. All experiments were done on machines with Intel Core 2 2.40GHz processor with 2 gigabytes of physical memory. The linear programs and integer programs on the data sets are solved using CPLEX Version 10.1.0. The Arya et al.'s single swap local search algorithm and Jain et al. algorithm are solved using MATLAB version 7.0. The tolerance for the bounded envelope that we use for the termination of binary search is 0.01 (see Lin et al. [9] for the bounded envelope procedure). For space reasons, we cannot present the full table of results; however, Figures 1 and 2 show how the costs of the k-median solutions from the integer optimum, Arya et al.'s local search algorithm, Charikar et al.'s LP rounding algorithm and the Jain et al.'s greedy algorithm compare to the linear program for different values of k for two sample datasets *pmed*40 and *Galvão150*. This performance was typical.

Even though the Arya et al.'s algorithm's performance guarantee is 5, in practice the local search algorithm performs much better than that. The local optimums are within 1% from the linear program optimum on average. Charikar's et al.'s LP rounding algorithm performs even better as most of the LP solutions are already integral or very close to being integral except for some small values of k. Note that the Jain et al.'s greedy LMP FL algorithm gives only a bounded envelope and does not give k-median solutions for all values of k. Here we can see that the LP rounding algorithm and the local search algorithm perform better than Jain et al.'s algorithm.

In terms of running time, the LP solver runs faster than the local search and greedy algorithm for all datasets. Also the IP solver takes a lot more time to solve all the instances of k for bigger datasets.

4.2 Incremental *k*-Median

In this section we compare the performances of four different incremental k-median algorithms on the selected datasets: Mettu and Plaxton's incremental

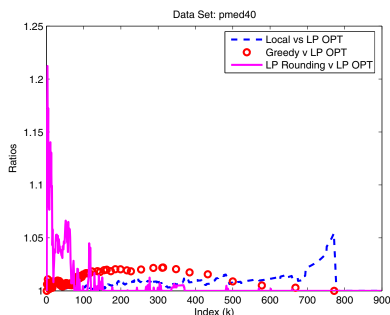


Fig. 1. Quality of solutions of k-median algorithms (dataset *pmed*40)

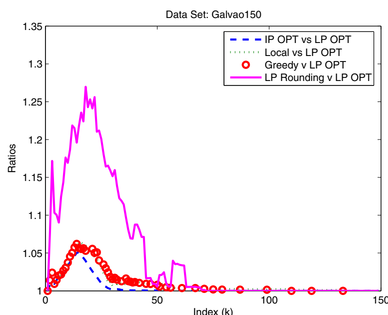


Fig. 2. Quality of solutions of k-median algorithms (dataset *Galvão150*)

k-median algorithm (MPInc), Lin et al.'s AltIncApprox algorithm with solutions from the Arya et al.'s single swap local search algorithm (LInc) and Charikar et al.'s LP rounding (LPR) and Lin et al.'s BoundedIncApprox algorithm with the bounded envelope obtained from the Jain et al.'s greedy LMP FL algorithm (GInc).

Our experiments show that Lin et al.'s algorithms perform much better than the Mettu and Plaxton's algorithm on the datasets. This inference is reinforced by Figures 3, 4, 5, and 6 which show that the ratios of the costs of solutions obtained from Lin et al.'s incremental algorithms to the LP optimum are always better than the corresponding ratios of Mettu and Plaxton's algorithm for a sample of datasets (*pmed*10, *pmed*25, *pmed*40 and *Galvão150*). The Mettu-Plaxton algorithm runs much faster than Lin et al.'s algorithms; these use a k-median algorithm or a bounded envelope algorithm as a blackbox, which make them very slow. However the quality of the incremental solutions obtained from Lin et al.'s algorithm is much better than that of the Mettu-Plaxton algorithm.

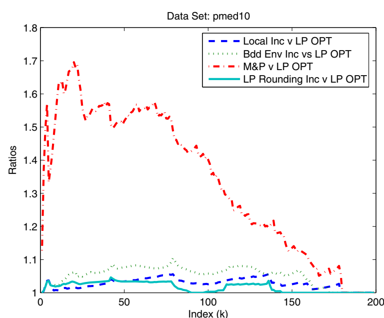


Fig. 3. Quality of solutions of incremental k-median algorithms (dataset *pmed*10)

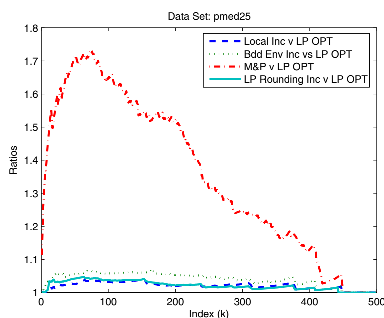


Fig. 4. Quality of solutions of incremental k-median algorithms (dataset *pmed*25)

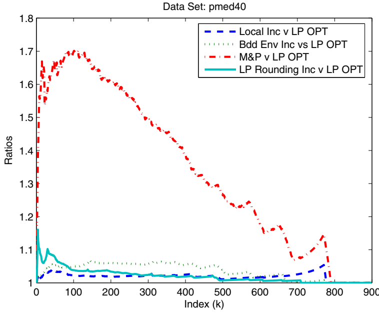


Fig. 5. Quality of solutions of incremental k-median algorithms (dataset *pmed*40)

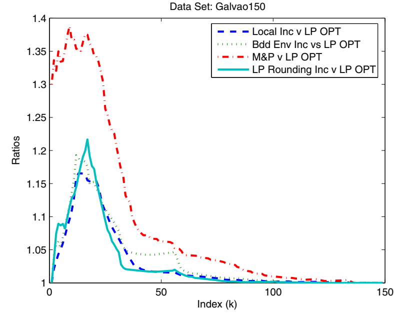


Fig. 6. Quality of solutions of incremental k-median algorithms (dataset *Galvão150*)

4.3 Hierarchical k-Median

In this section we compare the performance of Plaxton's hierarchical k-median algorithm against Lin et al.'s AltIncApprox hierarchical k-median algorithm on the datasets. Note that Plaxton's algorithm takes in any incremental k-median solution as input and outputs a parent function which defines the hierarchical solution. We give the incremental k-median solutions from our runs of AltIncApprox and the Mettu-Plaxton algorithms as input to the Plaxton's hierarchical algorithm (PHLI and PHMP) and compare them against Lin et al.'s hierarchical k-median algorithms' solutions (HL, HG and LPRH) for different datasets.

Figures 7, 8, 9, and 10 show how the costs of the hierarchical k-median solutions for different algorithms compare against the optimal linear program solutions for different values of k for sample datasets *pmed*10, *pmed*25, *pmed*40 and *Galvão150*. The algorithms we consider are AltIncApprox algorithm (using Arya et al.'s local search k-median solutions (HL) and Charikar et al.'s LP rounding solutions (LPRH)), BoundedIncApprox algorithm (using bounded envelope from Jain et al.'s greedy algorithm) (HG), Plaxton's hierarchical k-median algorithm on the incremental solutions of AltIncApprox algorithm (PHLI) and Plaxton's algorithm on Mettu and Plaxton's incremental k-median solutions (PHMP).

The hierarchical solutions obtained by AltIncApprox algorithms are better than other algorithms. The ratios for the PHMP algorithm are not as good as for the other algorithms since PHMP uses the incremental k-median solutions of Mettu and Plaxton as input which are not as good as other incremental algorithms in terms of quality. Lin et al.'s hierarchical algorithm (HL) which computes hierarchical solutions directly from k-median solutions performs better than the Plaxton's hierarchical algorithm even when the incremental solutions from AltIncApprox are given as input.

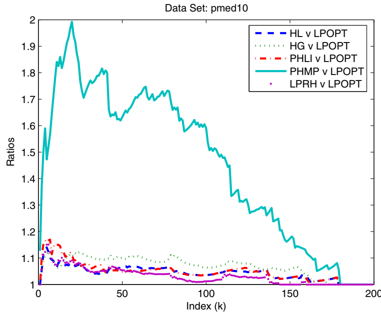


Fig. 7. Quality of solutions of hierarchical k-median algorithms (dataset *pmed*10)

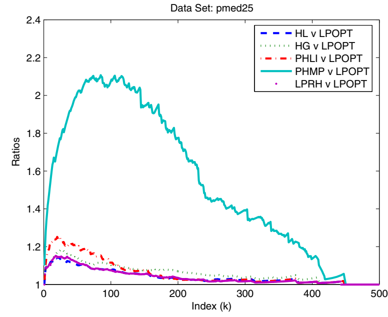


Fig. 8. Quality of solutions of hierarchical k-median algorithms (dataset *pmed*25)

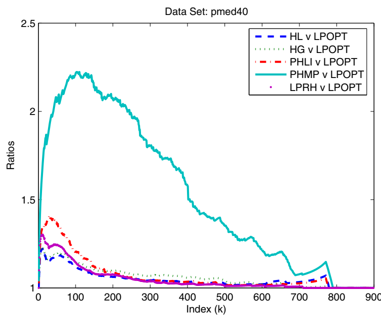


Fig. 9. Quality of solutions of hierarchical k-median algorithms (dataset *pmed*40)

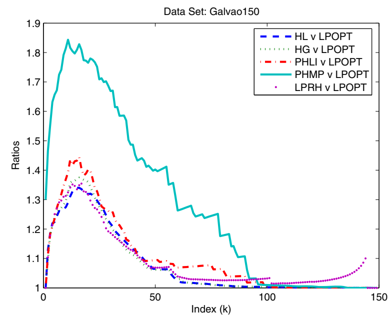


Fig. 10. Quality of solutions of hierarchical k-median algorithms (dataset *Galvão150*)

5 Conclusions

We evaluate different k-median, incremental k-median and hierarchical k-median algorithms on different datasets and show our results here. For the k-median problem, Charikar et al.'s LP rounding algorithm performs better and faster on average than other k-median algorithms like Arya et al.'s local search algorithm. We also notice that in many real-life datasets the optimal LP solution for the k-median problems for most values of k are integers which also makes the LP rounding techniques much better in terms of the quality of the solutions.

The quality of incremental solutions, when AltIncApprox algorithm is run on the k-median solutions of Arya et al.'s local search algorithm and Charikar et al.'s LP rounding algorithm, are much better than the incremental solutions of Mettu and Plaxton's algorithm. Even though the LP rounding algorithm performs poorly for some small values of k, Lin et al.'s incremental and hierarchical algorithms skips many of these poor solutions while bucketing the solutions

geometrically and this makes the corresponding incremental solutions comparable in quality to the incremental solutions obtained from Arya et al.'s local search k-median solutions.

Mettu and Plaxton's incremental k-median algorithm is much faster than the other incremental k-median algorithms we implement. However one important point to note here is that we find good k-median solutions for all values of k both in Arya et al.'s local search algorithm and Charikar et al.'s LP rounding algorithm. Most of these solutions are not used at all by the Lin et al. algorithms since it uses only one solution from each of the geometrically increasing buckets. It would be useful if we would somehow be able to find a sequence of k-median solutions that are geometrically increasing in cost in a faster way; this could lead to significant improvements in the running times of the Lin et al. algorithms.

References

1. Alp, O., Erkut, E., Drezner, D.: An efficient genetic algorithm for the p-median problem. Annals of Operations Research 122, 21–42 (22) (2003)
2. Arya, V., Garg, N., Khandekar, R., Meyerson, A., Munagala, K., Pandit, V.: Local search heuristics for k-median and facility location problems. SIAM Journal on Computing 33, 544–562 (2004)
3. Beasley, J.E.: A note on solving large p-median problems. European Journal of Operational Research 21, 270–273 (1985)
4. Charikar, M., Guha, S., Tardos, E., Shmoys, D.B.: A constant-factor approximation algorithm for the k-median problem (extended abstract). In: Proceedings of the 31th Annual ACM Symposium on Theory of Computing, pp. 1–10 (1999)
5. Chrobak, M., Kenyon, C., Noga, J., Young, N.E.: Incremental medians via online bidding. Algorithmica 50, 455–478 (2008)
6. Galvao, R.D., ReVelle, C.: A Lagrangean heuristic for the maximal covering location problem. European Journal of Operational Research 88(1), 114–123 (1996)
7. Jain, K., Mahdian, M., Markakis, E., Saberi, A., Vazirani, V.V.: Greedy facility location algorithms analyzed using dual-fitting with factor-revealing LP. Journal of the ACM 50, 795–824 (2003)
8. Kariv, O., Hakimi, S.L.: An algorithm approach to network location problems ii. the p-medians. SIAM Journal of Applied Mathematics 37, 539–560 (1979)
9. Lin, G., Nagarajan, C., Rajaraman, R., Williamson, D.P.: A general approach for incremental approximation and hierarchical clustering. SIAM Journal on Computing 39, 3633–3669 (2010)
10. Mettu, R.R., Plaxton, C.G.: The online median problem. SIAM Journal on Computing 32, 816–832 (2003)
11. Plaxton, C.G.: Approximation algorithms for hierarchical location problems. Journal of Computer and System Sciences 72, 425–443 (2006)

Engineering the Modulo Network Simplex Heuristic for the Periodic Timetabling Problem*

Marc Goerigk and Anita Schöbel

Institut für Numerische und Angewandte Mathematik
Georg-August-Universität Göttingen, Germany

Abstract. The *Periodic Event Scheduling Problem* (PESP), in which events have to be scheduled repeatedly over a given period, is a complex and well-known discrete problem with numerous real-world applications. One of them is to find periodic timetables which is economically important, but difficult to handle mathematically, since even finding a feasible solution to this problem is known to be NP-hard. On the other hand, there are recent achievements like the computation of the timetable of the Dutch railway system that impressively demonstrate the applicability and practicability of the mathematical model. In this paper we propose different approaches to improve the *modulo network simplex algorithm* [8], which is a powerful heuristic for the PESP problem, by exploiting improved search methods in the modulo simplex tableau and larger classes of cuts to escape from the many local optima. Numerical experiments on railway instances show that our algorithms are able to handle problems of the size of the German intercity railway network.

1 Introduction

The *Periodic Event Scheduling Problem* (PESP) as introduced in [12] models periodically reoccurring events that have to be scheduled according to given feasible time spans. Its general modeling power made it the model of choice for the computation of periodic timetables in public transport, see e.g. [5,9,7,3,10]. Recently, also connections to Graphical Diophantine Equations have been explored [2] in the case of multiple periods.

The applicability of the model to real-world problems has been impressively demonstrated by two recent milestones. In 2005, the new timetable for the underground railway of Berlin was introduced [4], being the first mathematically optimized railway timetable in practice. And in 2006 the largest Dutch railway company, the *Nederlandse Spoorwegen*, introduced a completely new timetable, with an estimated profit of 40 million Euro annually [1].

The most common approach to solving PESP problems is by mixed-integer programming techniques [6]. However, these approaches suffer from high computation times. In [8] a heuristic approach, the *modulo network simplex method*,

* Partially supported by grant SCHO 1140/3-1 within the DFG programme *Algorithm Engineering*.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 181–192, 2011.
© Springer-Verlag Berlin Heidelberg 2011

is presented, which is based on the classic network simplex method. To the best of our knowledge, this heuristic is currently the most powerful method to solve large instances.

The purpose of this paper is to improve the modulo network simplex method's performance for practical timetabling instances. We will show that by engineering the concept of the original method, we are able to compute solutions with both smaller runtimes and better objective values.

2 PESP and Periodic Timetabling

A *periodic event* i is a countably infinite set of events i_p, $p \in \mathbb{Z}$, with occurrence times

$$t(i_p) = t(i) + p \cdot T$$

for a given *period* T, see [12]. A *span constraint* consists of an interval $[l_{ij}, u_{ij}] \subset \mathbb{R}$ for a pair of events (i, j). The span constraint is satisfied if

$$(t(j) - t(i)) \bmod T \in [l_{ij}, u_{ij}].$$

The PESP problem is given as follows: For a given finite set of events with a period T and a finite set of span constraints, find a time $t(i)$ for each periodic event i such that all span constraints are satisfied. It is shown [12] that PESP is NP-hard by transformation from the Hamiltonian Circuit Problem.

Based on the PESP, the periodic timetabling problem can be formulated by introducing *Event-Activity-Networks* (EAN) to model the time-dependent behavior of the various vehicles considered [9]. EANs are directed graphs $G = (\mathcal{E}, \mathcal{A})$ with nodes

$$\mathcal{E} = \mathcal{E}_{arr} \cup \mathcal{E}_{dep}$$

that represent arrival and departure *events* of every train line at every station, and edges

$$\mathcal{A} = \mathcal{A}_{drive} \cup \mathcal{A}_{wait} \cup \mathcal{A}_{change} \cup \mathcal{A}_{head}$$

representing either driving, waiting and changing *activities* or the necessary security headway between vehicles sharing the same infrastructure. The events are periodic since all arrivals and departures are repeated in every period, and for each of the activities a span constraint is given which contains the minimal and the maximal duration of the activity. The minimal duration guarantees a certain level of robustness while the maximal duration controls the quality of the timetable. Figure 1 shows a small part of an EAN in which two trains share track capacities of the same station and are therefore connected by a headway activity. The orientation of such a headway activity is of no importance when the span constraint is chosen properly.

The goal is to find a timetable assigning a *time* $\pi_i := t(i) \mod T \in \mathbb{R}$ to each of the events $i \in \mathcal{E}$ for a given period T such that the span constraints are satisfied, i.e., $(\pi_j - \pi_i) \bmod T \in [l_{ij}, u_{ij}]$ for each activity $(i, j) \in \mathcal{A}$. The objective in the timetabling problem is not only to search for a *feasible* solution,

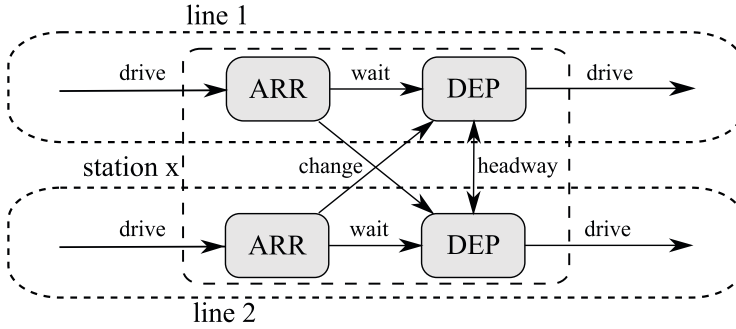


Fig. 1. Detail of an Event-Activity-Network

but instead for an *optimal* one, namely we minimize the total passenger traveling time given as

$$\sum_{(i,j)\in\mathcal{A}} (\pi_j - \pi_i) \mod T - l_{ij}.$$

Instead of the event times $\pi_i, i \in \mathcal{E}$, one can equivalently determine the *slack* $y_{ij} = \pi_j - \pi_i - l_{ij}$ for any edge $(i,j) \in \mathcal{A}$ with lower bound l_{ij}. Generally speaking, the slack of an activity is the amount of time spent additionally to its minimum duration. Using this concept, an alternative formulation (used by the modulo network simplex) has been suggested in [7]. Let $\mathcal{T} = (\mathcal{E}, \mathcal{A}_\mathcal{T})$ be a spanning tree with its corresponding fundamental cycle matrix Γ, then the *periodic timetabling problem* can be formulated as follows.

$$\begin{aligned}
\text{(PTT)} \qquad \min \; & \sum_{(i,j)\in\mathcal{A}} \omega_{ij} y_{ij} \\
\text{s.t} \quad & \Gamma(y+l) = Tz \\
& 0 \le y_{ij} \le u_{ij} - l_{ij} \; \forall (i,j) \in \mathcal{A} \\
& y_{ij} \in \mathbb{R} \; \forall (i,j) \in \mathcal{A} \\
& z_{ij} \in \mathbb{Z} \; \forall (i,j) \in \mathcal{A} \setminus \mathcal{A}_\mathcal{T},
\end{aligned}$$

where $y = (y_{ij})_{(i,j)\in\mathcal{A}}$ and $l = (l_{ij})_{(i,j)\in\mathcal{A}}$. For details and correctness we refer to [7,3]. As the variables z_{ij} model the periodic character of the problem, they will be referred to as *modulo parameters*.

Note that the modulo parameters are the reason why this problem is NP-hard. For fixed variables z_{ij} the timetabling problem is called aperiodic and is the dual of a minimum cost flow problem that can be solved efficiently using the network simplex method.

3 The Modulo Network Simplex Method

In this section we briefly describe the method of [8]. Its main idea is to encode a solution as a spanning tree $\mathcal{T}_l \cup \mathcal{T}_u$ by setting the modulo parameters of the tree

edges to 0 and the duration of these activities either to their respective lower or upper bound.

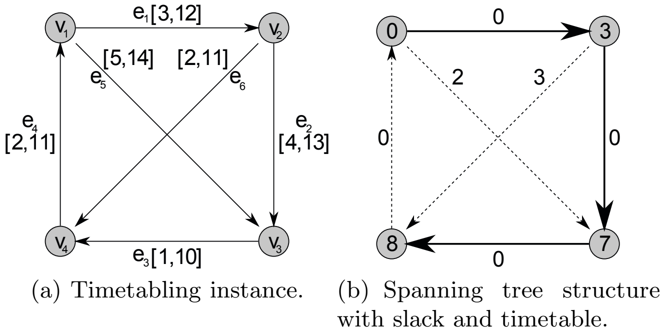
Definition 1. *[8] A* spanning tree structure $(\mathcal{T}_l, \mathcal{T}_u)$ *is a spanning tree* $\mathcal{T} = \mathcal{T}_l \cup \mathcal{T}_u$ *with an edge partition such that* y_{ij} *is set to* 0 *on all edges* $(i,j) \in \mathcal{T}_l$ *and set to* $u_{ij} - l_{ij}$ *for all edges* $(i,j) \in \mathcal{T}_u$.

A spanning tree structure uniquely determines a periodic timetable by calculating the slack y_{ij} for the missing edges $(i,j) \notin \mathcal{T}$ such that the cycle condition $\Gamma(y+l) = Tz$ of (PTT) holds. On the other hand, it is shown in [7] that

$$\binom{\pi}{z} \in \mathcal{Q} := \text{conv.hull}\left(\left\{\binom{\pi}{z} | l_{ij} \leq \pi_j - \pi_i + Tz_{ij} \leq u_{ij}; z \in \mathbb{Z}^m; \pi \in \mathbb{R}^n\right\}\right)$$

is an extreme point of $\mathcal{Q}$ if and only if it is a solution that is given by a spanning tree structure. Thus it is sufficient to investigate only these solutions.

The modulo network simplex works as follows: As it is the case in the classic network simplex method, a given feasible spanning tree solution is gradually improved by exchanging tree and non-tree edges that lie in the same fundamental cycle, i.e., the cycle that consists of the non-tree edge and its unique path in the spanning tree. This is done with the help of a simplex-like tableau.



(a) Timetabling instance. (b) Spanning tree structure with slack and timetable.

Fig. 2. A timetabling instance with a feasible spanning tree structure

Example 1. In Figure 2(a) a problem instance is given with period $T = 10$ and in 2(b) a feasible spanning tree structure, where $\mathcal{T} = \mathcal{T}_l$. The corresponding modulo simplex tableau can be seen in Table 3 (a). It contains the fundamental cycles that are induced by the non-tree arcs e_4, e_5 and e_6.

The objective value $w^t y$ is calculated by $\sum_{(i,j)\notin\mathcal{T}} w_{ij} y_{ij} + \sum_{(i,j)\in\mathcal{T}_u} w_{ij}(u_{ij} - l_{ij})$. Let y^{ij} be the slack vector after pivoting edges e_i and e_j. By writing $[y]_T := y \bmod T$ for short and denoting by b_{ij} the tableau entry for the edges e_i and e_j, the change in the objective value when pivoting a non-tree edge e_i and a tree edge e_j to $\mathcal{T}_l$ is

	e_1	e_2	e_3	e_4	e_5	e_6	y	ω
e_4	1	1	1	1	0	0	0	4
e_5	-1	-1	0	0	1	0	2	5
e_6	0	-1	-1	0	0	1	3	1
$\omega^t y$							13	

Fig. 3. The modulo simplex tableau associated with Figure 2(b)

$$\begin{aligned}
&\omega^t y^{ij} - \omega^t y \\
&= \sum_{k \in \mathcal{A} \setminus (\mathcal{T} \cup \{i\})} \omega_k \left[y_k - \frac{b_{kj}}{b_{ij}} y_i \right]_T + \omega_j \left[\frac{y_i}{b_{ij}} + y_j \right]_T + \sum_{k \in \mathcal{T}_u} \omega_k y_k - \sum_{k \in \mathcal{A}} \omega_k y_k \\
&= \sum_{k \in \mathcal{A} \setminus (\mathcal{T} \cup \{i\})} \omega_k \left(\left[y_k - \frac{b_{kj}}{b_{ij}} y_i \right]_T - y_k \right) + \omega_j \left(\left[\frac{y_i}{b_{ij}} + y_j \right]_T - y_j \right) - \omega_i y_i,
\end{aligned}$$

while the change when pivoting to T_u is calculated analogously.

Every non-zero entry of the left part of the table stands for a possible basis exchange of a non-tree-arc with a tree-arc that lies on its induced fundamental cycle. However, due to the modulo parameters, reduced costs as in the classic network simplex cannot simply be read off in the tableau. In consequence, the resulting change of every entry of the simplex tableau has to be calculated which results in a time-consuming complexity of $c \cdot (m - n + 1) \cdot (n - 1) \cdot (m - n + 1) = \mathcal{O}(m^2 n + n^3)$, where $m = |\mathcal{A}|$ and $n = |\mathcal{E}|$. Furthermore, as the problem is not convex, many local optima exist, which is the reason why methods of global optimization should be added.

In order to do so, we use the following correspondence between the pivoting operation in the modulo network simplex method and cuts, i.e., sets $\{(i,j) \in \mathcal{A} : i \in \mathcal{E}_1 \text{ and } j \in \mathcal{E}_2\} \cup \{(i,j) \in \mathcal{A} : i \in \mathcal{E}_2 \text{ and } j \in \mathcal{E}_1\}$ for a partition $\mathcal{E}_1 \dot\cup \mathcal{E}_2 = \mathcal{E}$. Every edge e of a spanning tree canonically induces a *fundamental* cut by taking the two connected components that appear when removing e. Pivoting a tree and a non-tree edge as it is done in the modulo network simplex method can therefore be interpreted as shifting slack from the edges of the corresponding fundamental cut to the non-tree edge. Thus, the modulo network simplex searches iteratively for improving fundamental cuts.

Notation. *Let a cut c be given by its node partition $\mathcal{E}_1 \dot\cup \mathcal{E}_2$, and let $\delta \in \mathbb{R}$. We say that we* apply *the cut c with δ, if the slack y_{ij} is increased by δ for all edges (i,j) with $i \in \mathcal{E}_1$, $j \in \mathcal{E}_2$ and decreased by δ for all edges (i,j) with $i \in \mathcal{E}_2$, $j \in \mathcal{E}_1$. Moreover, when the resulting modulo paramters are fixed and a new spanning tree structure is computed, the cut is called* globally improving*, if the objective value decreases.*

To overcome local optima, any other class of cuts can be chosen, which will force the full recomputation of the corresponding simplex-tableau. In that case,

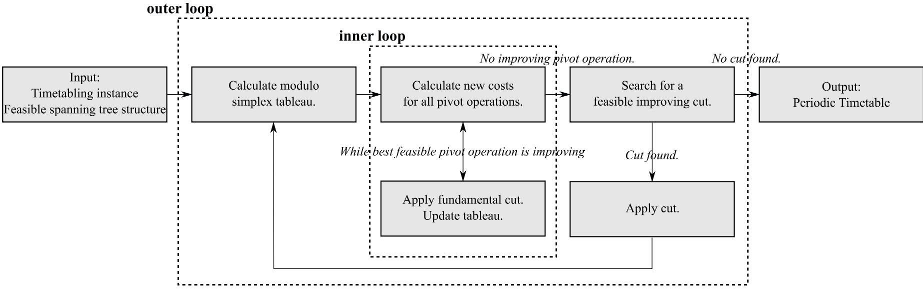


Fig. 4. Schematic process of the modulo network simplex

modulo parameters will have to be fixed and the dual min-cost balanced flow problem be solved in order to obtain a new spanning tree structure.

Figure 4 summarizes the main steps of the algorithm. The inner loop and outer loop shown will be used for our improvements in the next section.

4 Improving the Modulo Simplex

We proceed in two steps. On the one hand, we will improve the runtime of the algorithm by proposing alternative schemes for choosing a basis exchange pair in the inner loop, i.e., the fundamental cut. This is done in Section 4.1. On the other hand, in order to improve the quality of the solution, we will investigate several further cut possibilities that force the recomputation of the spanning tree structure used in the outer loop and hence overcome local optima, see Section 4.2.

4.1 Improving Runtimes: The Inner Loop

The time needed for investigating a single column of the simplex tableau grows quadratically in its number of non-zero entries. This is due to two reasons: The number of non-zeros determines the number of multiplications that have to be carried out for calculating the change of the objective function value, but also determines the number of possible basis exchanges that have to be tested. Since timetabling instances typically have sparse columns in the simplex tableau, we present two different algorithms that take advantage of this fact.

To illustrate this observation, we randomly created 100 simplex tableaus for a railway instance and counted the number of non-zeros for each column. The resulting distribution showed that about 10% of all columns contained more than 80% of all non-zero entries.

In terms of runtimes, these are the least effective columns to check. Thus, by neglecting only 10% of all tree-edges, the largest part of computation time is saved. This is exploited by the following two algorithms:

Modus "First Quality": Search the columns of the modulo simplex tableau for an improving pivot operation beginning from the one with fewest non-zeros. As big improvements are possible in the beginning of the loop, but only small ones

at the end, a dynamically updated criterion is used which stops the procedure if the change in the objective value is considered satisfactory.

Modus "Percentage": Search only the $p\%$ of the columns, namely those with the fewest non-zero entries, for an improving pivot operation and apply the best pivot operation change found. Some randomly created simplex tableaus can be used to determine how large the search space should be and hence to fit the parameter p to the particular timetabling instance. Choosing p is an easy possibility to decide on the time the algorithm may use for the inner loop.

Both approaches avoid searching the whole simplex tableau while neglecting as few pivot possibilities as possible.

4.2 Improving Quality: The Outer Loop

To escape a local minimum of the outer loop, we investigate four classes of non-fundamental cuts. The first one, single node cuts, has been suggested in [8] while the other three are new.

Single Node Cuts. The time π_i of a single event i is delayed by $\delta \in \mathbb{Z}$, i.e., the slack y_{ji} of in-going edges (j, i) is increased by δ, while slack of outgoing edges (i, j) is decreased.

In [8], a single node cut is applied, if

$$\sum_{(i,j)\in\mathcal{A}\,:\,i\in\mathcal{E}_1,j\in\mathcal{E}_2} \omega_{ij}(y_{ij}-\delta) + \sum_{(j,i)\in\mathcal{A}\,:\,i\in\mathcal{E}_1,j\in\mathcal{E}_2} \omega_{ji}(y_{ji}+\delta) < 0, \qquad (1)$$

where $\mathcal{E}_1 = \{i\}$, $\mathcal{E}_2 = \mathcal{E} \setminus \{i\}$ for a node $i \in \mathcal{E}$. This is based on the obervation that whenever the *local* condition (1) holds, the cut is *globally* improving. We will call cuts fulfilling (1) *locally improving*. However, as a given spanning tree structure is optimal with respect to the induced modulo parameters, a necessary condition for a single node cut to be locally improving is that it changes at least one modulo parameter. This is unlikely to happen, taken into consideration that the slack y_{ij} of at least one of the adjacent edges is set to be 0 or $u_{ij} - l_{ij}$, as the solution is induced by a spanning tree structure.

Waiting Edge Cuts. To improve the probability of finding a single node cut that is feasible, i.e., respects the time constraints $0 \le y_{ij} \le u_{ij} - l_{ij}$ for all $(i, j) \in \mathcal{A}$, we consider cuts which are induced by an edge (i, j) with a small feasible time span $u_{ij} - l_{ij}$. By doing so, the slack of the edge (i, j) does not need to be changed, thus increasing the probability that the cut is feasible. Here we use another characteristic of timetabling instances, namely that activities with small time spans $u_{ij} - l_{ij}$ are usually the activities of $\mathcal{A}_{\text{wait}}$. We hence denote the class of cuts that are induced by a partition

$$\{i, j\} \cup \mathcal{E} \setminus \{i, j\} \text{ for } (i, j) \in \mathcal{A}_{\text{wait}}$$

as *waiting edge cuts.*

Random Node Cuts. (1) is only a sufficient condition, i.e., cuts may still be globally improving, when they are not locally improving. In our method of random node cuts we apply feasible single node cuts, neglecting if (1) holds or not, i.e., whether they improve locally. Hence we force the inner loop of the modulo network simplex algorithm to start again, even at the cost of a possible temporary higher objective value.

Multi Node Cuts. The next approach investigates the class of cuts which are created by iteratively expanding single node cuts. Let a cut c that is induced by the partition $V_1 \dot\cup V_2$ and a single node cut c_i for node $i \in V_2$ be given in a directed graph $G(V, E)$. Then we call the cut that is induced by $(V_1 \cup \{i\})$ and $(V_2 \setminus \{i\})$ the *exclusive union* of these cuts and denote it by $c \oplus c_i$. The following lemma shows that *all* cuts can be written by using single node cuts.

Lemma 1. *Let a cut c in a graph $G(V, E)$ be given by its node partition $V_1 \dot\cup V_2$. Then c can be written as the exclusive union of single node cuts: $\bigoplus_{i \in V_1} c_i = c$.*

Proof.
First, let $e \in c$. Then there holds either $e = (i, j)$ or $e = (j, i)$ with $i \in V_1$ and $j \in V_2$. As there is a single node cut for node i, namely c_i, but by construction none for node j, e is contained in the exclusive union of the single node cuts.

Now let $e \in \oplus c_i$. By construction of the single node cuts again either source or target of the edge is in V_1, while the other node is in V_2. Therefore, $e \in c$. □

As a direct consequence, the change of a timetabling solution by applying any cut with a δ is equivalent to the successive application of single node cuts with this same δ. In fact, both approaches of defining a cut by its node partition or by its single node cuts are equivalent, as the following theorem states.

Theorem 1. *Let $G(V, E)$ be a connected graph and $c = \oplus_{i \in I} c_i$ a nonempty cut, given by single node cuts and induced by the partition $V_1 \dot\cup V_2$. Then $I = V_1$ (or $I = V_2$ if the orientation of the cut is neglected).*

Proof.
- First, let $V_1 \subsetneq I$. Assume that there are adjacent nodes $i, j \in I$ with $i \in V_1$ und $j \notin V_1$. Then $(i, j) \notin \oplus_{i \in I} c_i$, but $(i, j) \in c$ - therefore, such adjacent nodes cannot exist. As c is nonempty, $I = V$ cannot hold. As G is connected, there is a node $i \in I \setminus V_1$ that is adjacent to a node $j \notin I$. Therefore the edge (i, j) would be contained in $\oplus_{i \in I} c_i$, but not in c.
- Now, let $I \subsetneq V_1$. Then there holds $\oplus_{i \in I} c_i = \oplus_{j \in V \setminus I} c_j$ up to orientation and this case is reducible to $V_2 \subsetneq J$ with $J = V \setminus I$, which has already been considered.
- Finally, let $I \cap V_1 \neq \emptyset$, $I \cap V_2 \neq \emptyset$ and I no superset of V_1 or V_2. To have $c = \oplus_{i \in I} c_i$, for every edge (i, j) with $i \in V_1$ and $j \in V_2$ w.l.o.g. there has to hold $i \in I$ and $j \notin I$. From the assumptions, there is $i \in V_2 \cap I$ and $j \in V_2 \setminus I$. As the graph is connected, there is a path from i to j. i cannot be adjacent to a node in V_1, as then $i \in I$ could not hold. There fore there is an edge (x, y) with $x \in V_2 \cap I$ and $y \in V_2 \setminus I$, that is contained in $\oplus_{i \in I} c_i$, but not in c. □

We will call a cut that is given by the node partition $V_1 \dot\cup V_2$ *connected*, if both subgraphs $G_1(V_1, E_{V_1})$ and $G_2(V_2, E_{V_2})$ are connected. We can state the following criterion for the connectivity of a cut.

Corollary 1. *A cut $c = \oplus_{i \in I} c_i$ in a graph $G(V, E)$, where each of the c_i is a single node cut, is connected if and only if the subgraphs that are induced by I and $V \setminus I$ are connected.*

Similar to connected components in graphs, we can define connected components of cuts. Let a cut c induced by the node partition $V_1 \dot\cup V_2$ and a subcut $c' \subset c$, induced by $V_1' \dot\cup V_2'$ with $V_1' \subset V_1$ and $V_2 \subset V_2'$, be given. c' is called *connected component* of c, if c' is connected and there is no cut c'' with $c' \subsetneq c'' \subset c$, that is connected as well. We can easily conclude the following property:

Theorem 2. *Connected components of a cut are pairwise disjoint.*

Proof.
Let c^1 and c^2 be connected components of a cut c with $c^1 \cap c^2 \neq \emptyset$. Then $c' := c^1 \cup c^2$ is connected and $c^1 \subsetneq c' \subset c$. By the definition of connected components, c^1 and c^2 cannot share a common edge. $\square$

Therefore, if a cut is applied to a timetabling solution, then the objective value changes according to the sum of changes on every single edge. As connected components are disjoint, the objective value changes according to the sum of changes in every component.

Corollary 2. *In a timetabling solution the change of the objective value by applying a cut c with $\delta \in [0, T-1]$ equals the sum of changes of the connected components.*

This means that the search for a locally improving cut in an Event-Activity-Network can be *restricted to connected cuts.* This result is exploited by a greedy search algorithm, which successively enlarges a set of nodes starting from a randomly chosen single node i until the induced cut is locally improving. The algorithm therefore restricts its search space to connected cuts. We refer to the resulting cuts as *multi node cuts.*

5 Experiments

In this section the performance of the proposed improvement techniques is evaluated. For our experiments we used close to real-world instances of the *LinTim* toolbox [11]. All calculations were carried out on machines with 64 Bit Dual Core AMD Opteron Processors running at 2000 MHz with 12 GB working memory. The average need for RAM was only at about 50 MB. Table 1 gives an overview about the sizes of the five instances we considered.

To find a feasible spanning tree structure we applied a constraint propagation approach that is able to find a feasible solution within some minutes of computation time. The modulo parameters found are then fixed and the dual problem

Table 1. Instance sizes

Instance	Events	Activities
Small	3533	5575
Medium 1	3664	6378
Medium 2	3668	6556
Large 1	4184	7061
Large 2	4873	7898

Table 2. Objective value improvement in percent with respect to initial solution using steepest descent. The best result per row is written in bold font.

Instance	Single Node Cuts	Waiting Edge Cuts	Random Cuts	Multi Node Cuts
Small	17	18	21	**28**
Medium 1	24	20	24	**29**
Medium 2	17	17	20	**29**
Large 1	23	23	27	**34**
Large 2	20	15	26	**32**

is solved with the help of the classic network simplex method. The resulting underlying spanning tree structure of the network simplex is then used as input for the modulo network simplex.

We evaluate the possible combinations between search methods for the inner and the outer loop of Figure 4 in the following ways:

1. For the inner loop, we use the steepest descent method. This is the original choice of the modulo network simplex. To escape local optima in the outer loop, we either use single node cuts, waiting edge cuts, random cuts or multi node cuts.
2. We fix the usage of single node cuts as the search method of the outer loop, as in the original method. For the inner loop, we use steepest descent, modus percentage and modus first quality.

For the first set of experiments, Table 2 gives an overview about the relative improvement of the initial solution. In the case of single node, multi node and waiting edge cuts, the search was performed until no more feasible and improving cut was found. This cannot be applied to random cuts - we hence restricted the number of random node cuts to three.

On average, using single node cuts improves the objective value of the initial solution by 20.2%, while the usage of multi node cuts yields an average improvement of 30.4%. For every single instance, the multi node cuts performed best, with a difference of up to 12%. Waiting edge cuts perform similar to single node cuts. In spite of their simplicity, random cuts yield surprisingly good results.

Concerning the second set of experiments, Table 3 compares the runtimes in seconds. Modus first quality and modus percentage outperform steepest descent on every instance.

Table 3. Runtimes in seconds using single node cuts. The best result per row is written in bold font.

Instance	Steepest Descent	Modus First Quality	Modus Percentage
Small	1747	956	**895**
Medium 1	4223	**1661**	1727
Medium 2	3385	**1531**	1660
Large 1	4878	2409	**1464**
Large 2	6685	3575	**3233**

Table 4. Objective value improvement in percent with respect to initial solution using single node cuts. The best result per row is written in bold font.

Instance	Steepest Descent	Modus First Quality	Modus Percentage
Small	17	20	**21**
Medium 1	24	22	**25**
Medium 2	17	19	**19**
Large 1	23	**26**	22
Large 2	20	26	**29**

The results show that the runtimes are improved by up to 70% in the case of *Large 1*, with average runtimes of 4184 seconds for steepest descent, 2026 seconds for modus first quality and only 1799 seconds for modus percentage.

Of course, achieving small runtimes with high objective values would be a Pyrrhic victory - in fact, the "fastest algorithm" would be to do nothing, i.e., not to improve the given solution at all. We therefore show the respective solution quality in Table 4.

It can be seen that the modi percentage and first quality significantly improve the calculation times compared to the method from [8], while being competitive in quality.

6 Conclusion

We have analyzed a powerful method for solving the PESP in the case of timetabling instances. Specific problem properties have been exploited and used to improve the runtime per iteration as well as avoid getting stuck in a local minimum. The superiority of some of the possible combinations of these approaches was demonstrated on timetabling instances.

Further research includes using the modulo network simplex method for *robustness* purposes. Basically, reducing the average slack results in timetables that have shorter traveling times for passengers, but also less buffer times and thus are more sensitive to disruptions. When using the neighborhood search

as a "black box" model for other objective functions like a preferable distribution of buffer times, we will be able to use the presented methods to create solutions that can cope better with unavoidable disruptions.

References

1. Kroon, L., Huisman, D., Abbink, E., Fioole, P., Fischetti, M., Marti, G., Schrijver, A., Steenbeek, A., Ybema, R.: The new dutch timetable: The OR revolution. Interfaces 39, 6–17 (2009)
2. Galli, L., Stiller, S.: Strong formulations for the multi-module PESP and a quadratic algorithm for graphical diophantine equation systems. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6346, pp. 338–349. Springer, Heidelberg (2010)
3. Liebchen, C.: Periodic Timetable Optimization in Public Transport. PhD thesis, Technische Universität Berlin (2006)
4. Liebchen, C.: The first optimized railway timetable in practice. Transportation Science 42(4), 420–435 (2008)
5. Liebchen, C., Möhring, R.: The modeling power of the periodic event scheduling problem: railway timetables — and beyond. In: Geraets, F., Kroon, L.G., Schoebel, A., Wagner, D., Zaroliagis, C.D. (eds.) Railway Optimization 2004. LNCS, vol. 4359, pp. 3–40. Springer, Heidelberg (2007)
6. Liebchen, C., Proksch, M., Wagner, F.H.: Performances of algorithms for periodic timetable optimization. In: Computer-aided Systems in Public Transport, pp. 151–180. Springer, Heidelberg (2008)
7. Nachtigall, K.: Periodic Network Optimization and Fixed Interval Timetables. Habilitationsschrift, Deutsches Zentrum für Luft- und Raumfahrt Braunschweig (1998)
8. Nachtigall, K., Opitz, J.: Solving periodic timetable optimisation problems by modulo simplex calculations. In: Proc. ATMOS (2008)
9. Odijk, M.: A constraint generation algorithm for the construction of periodic railway timetables. Transportation Research (B) 30, 455–464 (1996)
10. Peeters, L.: Cyclic Railway Timetable Optimization. PhD thesis, Erasmus University of Rotterdam (2003)
11. Schachtebeck, M., Schöbel, A.: Lintim – a toolbox for the experimental evaluation of the interaction of different planning stages in public transportation. Technical report, ARRIVAL Report 206 (2009)
12. Serafini, P., Ukovich, W.: A mathematical model for periodic scheduling problems. SIAM J. Disc. Math., 550–581 (1989)

Practical Compressed Document Retrieval⋆

Gonzalo Navarro[1], Simon J. Puglisi[2], and Daniel Valenzuela[1]

[1] Dept. of Computer Science, University of Chile,
{gnavarro,dvalenzu}@dcc.uchile.cl
[2] School of Computer Science and Information Technology
Royal Melbourne Institute of Technology,
simon.puglisi@rmit.edu.au

Abstract. Recent research on document retrieval for general texts has established the virtues of explicitly representing the so-called *document array*, which stores the document each pointer of the suffix array belongs to. While it makes document retrieval faster, this array occupies a significative amount of redundant space and is not easily compressible. In this paper we present the first practical proposal to compress the document array. We show that the resulting structure is significatively smaller than the uncompressed counterpart, and than alternatives to the document array proposed in the literature. We also compare various known algorithms for document listing and top-k retrieval, and find that the most useful combinations of algorithms run over our new compressed document arrays.

1 Introduction

Document retrieval queries aim at finding the documents of a text collection most relevant to a given query, where relevance is usually defined on frequency grounds. Such queries have been classically privative of Natural Language collections and handled with inverted indexes. In the last decade, however, there have been various research efforts towards generalizing them to arbitrary text collections, where the texts can correspond to ADN or protein sequences, text in Oriental languages (some of which cannot be easily split into words), program code, and symbolic sequences in general. See Gagie et al. [7] for a recent survey.

Muthukrishnan [16] established important milestones in this area. He proposed, among other less popular ones, the following fundamental document retrieval queries, which form the basis of more sophisticated retrieval activities:

- *Document listing:* List the *ndoc* distinct documents where a pattern p appears as a substring.
- *Frequency computation:* Same as above but also compute the number of times p appears within each returned document.
- *Top-k retrieval:* Find the k documents where p appears most often.

⋆ Partially funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile; by Fondecyt Grant 1-110066, Chile; and by the Australian Research Council.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 193–205, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Assume the text collection is formed by n documents, which are strings over alphabet $[1, \sigma]$, and let us call $T[1, N]$ their concatenation. Classical text indexes [1,14] can find *all* the *occ* occurrences of pattern p in T in time $O(|p| + occ)$ and then filter the *ndoc* distinct documents. This solves document listing, but *occ* can be much larger than *ndoc*. Muthukrishnan [16] showed how to solve the document listing query in $O(|p| + ndoc)$ time, which is essentially optimal.

A serious concern on this solution is space, however. It requires $O(N \lg N)$ bits, much more than the $N \lg \sigma$ bits required by the text itself. Part of this space is used by a suffix tree [1], which can be easily replaced by one of the many compressed suffix arrays (CSAs) of the literature [17,4]. Such CSAs represent a suffix array [14] plus the text, all within compressed space $|\mathsf{CSA}| \leq N \lg \sigma$.

The other part of the space owes to a so-called *document array*. Much research has been carried out around the problem of representing it in compact form [22,20,11,8,3]. The current situation is that one either has to spend $N \lg n$ bits for representing the document array using a *wavelet tree* [10], or one can simulate it using just $o(N)$ bits on top of $|\mathsf{CSA}|$. The second choice is clearly preferable both in theory and in practice for document listing.

However, document listing is just the most elementary activity. In order to rank documents by importance to the query, frequency computation and top-k retrieval are essential, and in this case the situation is very different. The wavelet tree representation directly computes frequencies and supports heuristics for top-k queries. The alternative simulation requires now the space for the global CSA *and* the CSA of each individual document. This turns out to be slower and require much more space in practice than the wavelet-tree based solution [3].

Therefore, the current status is that, if we wish not only to carry out document listing, but also to report frequencies, or to do top-k retrieval, the best alternative in practice is to store a wavelet tree representing the document array, which requires $N \lg n$ bits. This is possibly much more than the (at most) $N \lg \sigma$ bits required by a CSA (which by itself represents T and offers classical indexed text searches). The known techniques to compress wavelet trees [10] do not work for the document array.

In this paper we introduce the first compressed representation of wavelet trees that is useful for the document array. The representation uses grammar compression (precisely, RePair [12]) to exploit repetitions in the array. Such repetitions arise as a consequence of the compressibility of the text collection [9,7]. Our experiments over various collections show that our technique obtains significant space reductions, up to 40% of the original wavelet tree sizes. In exchange, operation times are higher. Yet we confirm that the time (and space) is still better than the alternative of not using wavelet trees for the problem of document listing with frequencies. We also study the wavelet trees in combination with various techniques for top-k retrieval [11,3], where our compressed wavelet trees offer a very attractive space/time tradeoff.

Our representation might have independent interest, as it is the first in grammar-compressing a sequence while supporting symbol *rank* and *select* operations on it. Those operations are useful in a wealth of applications.

2 Related Work

Muthukrishnan's solution [16] made use of the so-called *document array*. A *suffix array* $A[1,N]$ points to all the suffixes of $T[1,N]$ in lexicographic order [14]. All the occurrences of any p in T are pointed from an interval $A[sp,ep]$, which can be found in time $O(|p|\lg N)$ (or $O(|p|)$ with the help of the suffix tree [1]). The document array $D[1,N]$ is such that $D[i]$ tells the document suffix $A[i]$ belongs in T. So the document listing problem is solved by first finding $[sp,ep]$ using A, and then listing the distinct values in $D[sp,ep]$. To do this in $O(ndoc)$ time, Muthukrishnan uses a second array $C[1,N]$, which at $C[i]$ stores the last $j < i$ such that $D[j] = D[i]$. C must also answer range minimum queries (RMQs), telling in constant time the position of the minimum in a range of C cells.

In order to reduce space, the suffix tree or array can be replaced by a compressed suffix array (CSA) [17,4], which stores both A and T in compressed space, for example within $|\mathsf{CSA}| = (1+\frac{1}{\epsilon})H_0(T) + o(n\lg\sigma)$ bits [19] or $|\mathsf{CSA}| = nH_k(T) + o(n\lg\sigma)$ bits [5,10], where $H_k(T) \le \lg\sigma$ is the empirical k-th order entropy of T [17]. CSAs find $[sp,ep]$ in time as good as $search(p) = O(|p|\lg N)$ [19] or even $search(p) = O(|p|\lg\sigma)$ [5,10]. They compute any $A[i]$ or $A^{-1}[i]$ value in time t_A, for example $t_A = O(\log^{\epsilon} N)$ [19] or $t_A = O(\log^{1+\epsilon} N)$ [5,10]. They can also reproduce any text substring.

The document array D, however, requires $N\lg n$ bits, which is significant and totally redundant: one can infer $D[i]$ from $A[i]$ and some information on the limits of the documents in T. Array C is even more space-consuming, $N\lg N$ bits, and equally redundant. The RMQ data structure [6] adds $2N + o(N)$ bits. This extra space limits the applicability of the solution to document retrieval.

There have been various approaches to reduce the space of Muthukrishnan's solution. Mäkinen and Välimäki [22] used a *wavelet tree* [10] to represent D. While the wavelet tree takes essentially the same space of the plain representation, $N\lg n + o(N\lg n)$ bits, it can emulate array C, which is thus not represented. The time for document listing becomes $O(search(p) + ndoc\lg n)$. The wavelet tree also allowed them to compute the frequency of p within any document, in time $O(\lg n)$. The RMQ data structure was still necessary.

Gagie et al. [8] showed that the wavelet tree was powerful enough to get rid of the whole Muthukrishnan's algorithm. The wavelet tree alone, through a so-called *range quantile* operation, was able to deliver the distinct elements in $D[sp,ep]$, with their frequencies, in $O(\lg n)$ time per delivered item.

Culpepper et al. [3] explored different heuristics to solve the top-k problem using this very same wavelet tree. They found out that their so-called "greedy" heuristic was able to find the top-k documents much faster than listing them all and choosing the most frequent ones. They also showed that the structure was competitive with inverted indexes on Natural Language text collections.

A parallel development started with Sadakane [20]. He proposed to store a bitmap $B[1,N]$ marking with a 1 the positions in T where the documents started. B was enhanced with *rank* operations: $rank(B,i)$ tells the number of 1s in $B[1,i]$. Hence $D[i] = rank(B,A[i])$ could be computed with very little extra space on top of the CSA: A compressed representation of B requires just

$n \lg \frac{N}{n} + O(n) + o(N)$ bits and supports *rank* in constant time [18]. To emulate Muthukrishnan's algorithm, Sadakane showed that access to C is not really needed, just RMQ queries on C. He designed an RMQ structure using $4N+o(N)$ bits that does not need to access C. The time to list each document is $O(t_A)$. Both in theory and in practice, this solution is competitive in time and uses much less space than those based on wavelet trees, yet it only solves document listing. Hon et al. [11] showed how to reduce the extra space to just $o(N)$ by sparsifying the RMQ structure, yet the time raises to $O(t_A \lg^{1+\epsilon} N)$.

For computing frequencies, Sadakane [20] proposed to store a CSA for each document d of the collection. By computing A and A^{-1} a constant number of times over the global CSA and that of document d, it is possible to compute frequencies on document d. An extra, symmetric, RMQ data structure must be stored for this sake. Thus the space is $2|\mathsf{CSA}| + O(N)$ bits, which may compare favorably to the $|\mathsf{CSA}| + N \lg n + o(N \lg n)$ bits needed by wavelet trees. The time for computing a frequency is $O(t_A)$, which again may compare favorably with wavelet tree's $O(\lg n)$. In practice, however, Culpepper et al. [3] found that many small CSAs posed a much higher space overhead than that of the global CSA, so the structure was much larger than the wavelet tree. The speed was also slower than that offered by wavelet trees. We confirm in this paper that the solution is still slower than our slower-and-smaller compressed wavelet trees.

Hon et al. [11] showed that the second RMQ data structure introduced by Sadakane is unnecessary if one accepts an $O(\lg N)$ slowdown factor. In the light of the results of Culpepper et al. [3], this is unlikely to change matters in practice, because a reduction in $2N$ bits is insignificant but the slowdown is not. The key contribution of Hon et al., however, is an algorithm for top-k retrieval with time guarantees (which the heuristics of Culpepper et al. do not offer). Hon et al. build a sparse suffix tree on the collection, so that top-k queries over an interval of multiples of $g = k \cdot b$, for a parameter b, are precomputed. Thus to solve for any interval $[sp, ep]$, only $O(kb)$ elements at the extremes must be accessed, their frequency counted, and possibly inserted into the precomputed result. The extra space is $O((N/b) \lg N \lg n)$ bits on top of a solution for computing frequencies. By choosing $b = \Theta(\lg^{2+\epsilon} N)$, this space is $o(n)$ and the time for top-k queries is $O(k\, t_A \lg^{3+\epsilon} N)$. Any of the discussed solutions for computing frequencies can be used, and thus wavelet trees are of interest as a building block of this solution. There is no comparison in the literature, however, between this technique and the heuristics of Culpepper et al. [3], which as explained work on wavelet trees.

Thus, the best performance in practice is given by the wavelet tree of D, but its space is still high. The only clue at compressing it was given by Gagie et al. [7], who noted that D contains almost the same repetitions of the *differential* suffix array [9], and thus a grammar-based compression would reduce its size when the text is compressible. The practical impact of this theoretical result had not been verified, however. Moreover, the situation is more complicated because we do not need to represent D, but the wavelet tree of D, in order to support the various document retrieval tasks. The main point of this paper is to implement a grammar-compressed wavelet tree for D and evaluate its practical performance.

3 Bitmaps and Wavelet Trees

Given a bitmap $B[1, N]$, we define, for $b \in \{0, 1\}$, operation $rank_b(B, i)$ as the number of occurrences of bit b in $B[1, i]$, and $select_b(B, j)$ as the position in B of the jth occurrence of bit b.

Both operations can be solved in constant time by spending $o(N)$ bits on top of B [15], or by representing B in compressed form [18]: Let m be the number of 1s in B, then the total space is $m \lg \frac{N}{m} + O(m) + o(N)$.

A *wavelet tree* [10] represents a sequence $S[1, N]$ over an alphabet $[1, \rho]$. At the root it stores a bitmap $B[1, N]$ so that $B[i] = 0$ iff $S[i] \leq \rho/2$. The left child of the root represents the subsequence of S formed by the symbols $\leq \rho/2$; the other symbols form a subsequence at the right child. Those children are processed recursively over their alphabet ranges, until reaching the leaves. The wavelet tree has $O(\rho)$ nodes and height $\lceil \lg \rho \rceil$. Its bitmaps add up $N \lceil \lg \rho \rceil$ bits.

By using *rank* and *select* on the bitmaps, the wavelet tree gives access to any $S[i]$ in time $O(\lg \rho)$, thus it constitutes an alternative representation of S within about the same size, $N \lg \rho + o(N \lg \rho)$ bits. Other wavelet tree traversals compute also symbol *rank* and *select* on S, where the argument b can be any $c \in [1, \rho]$, also in time $O(\lg \rho)$. As explained, the wavelet tree is also useful for other types of queries of interest (in particular) to document retrieval [8,3].

If the compressed bitmap representation is used [18], then the space of the wavelet tree becomes the zero-order entropy of S, $NH_0(S) + o(N \log \sigma)$ [10]. In our case, however, the zero-order entropy of the document array is likely to be $\lg n$ bits per symbol, unless the document sizes are very different. There is no relation to the compressibility of the text itself.

4 Grammar Compression of Bitmaps

We describe a grammar-based compression of bitmaps $B[1, N]$ that supports *rank* and *select* operations on the compressed representation. We focus on Re-Pair [12] compressor. It successively finds the most frequent pair of symbols in the text, yz, and replaces it by a new symbol x (which can be involved in further pairings), adding a new grammar rule $x \rightarrow yz$. When all the pairs are unique, RePair terminates and delivers the remaining sequence, $\mathcal{C}$, and the set of rules, $\mathcal{R}$. We use a variant that generates a balanced grammar [21], of height $O(\lg N)$.

For providing random access we use sampling. Let $\ell(c) = 1$ for terminals c, and for nonterminals let $\ell(x)$ be the length of the string of terminals x expands to (that is, $\ell(x) = \ell(y) + \ell(z)$ if $x \rightarrow yz \in \mathcal{R}$). Now let $L(i) = 1 + \sum_{j=1}^{i-1} \ell(\mathcal{C}[j])$ the starting position in B of the symbol $\mathcal{C}[i]$ when expanded.

We sample B at regular intervals s. For each position $B[i \cdot s]$ we store $P[i] = (p, o, r)$, where p is the position in $\mathcal{C}$ of the symbol whose expansion will contain $B[i \cdot s]$, that is, $p = \max\{j,\ L(j) \leq i \cdot s\}$. The second component is the offset within that symbol, $o = i \cdot s - L(p)$, and the third is the rank up to that symbol, $r = rank_1(B, L(p) - 1)$. Finally, we store, for the nonterminals x, the length $\ell(x)$ and the number of 1s, $r(x)$, of the string of terminals they expand to.

To answer $rank_1(B,i)$, we compute $j = \lfloor i/s \rfloor$ and $P[j] = (p,o,r)$. We then start from $\mathcal{C}[p]$ with position $l = L(p) = i - o$ and rank r. From position p we advance in $\mathcal{C}$ until $l > i$. Each symbol of $\mathcal{C}$ can be processed in constant time while l and r are updated, since we know $\ell(x)$ and $r(x)$ for any symbol x. Finally we arrive at a position $p' \geq p$ so that $l = L(p') \leq i < L(p'+1) = l + \ell(\mathcal{C}[p'])$. At this point we complete our computation by recursively expanding $\mathcal{C}[p'] = x$. Let $x \rightarrow yz \in \mathcal{R}$, then if $l + \ell(y) \leq i$ we expand y; otherwise we increase l by $\ell(y)$, r by $r(y)$, and expand z. As the grammar is balanced the total time is $O(s + \lg N)$.

For *select* we obtain the same complexity by first binary searching P to find the right interval and then traversing sequentially the block, until exceeding the desired number of 0s or 1s, and finally expanding the last symbol of $\mathcal{C}$.

Let $R = |\mathcal{R}|$ be the number of rules in the grammar and $C = |\mathcal{C}|$ the length of the final array. Then a RePair compressor would require $O((R+C)\lg R)$ bits. Our representation requires $O(R \lg N + C \lg R + (N/s)\lg N)$, and the time for the operations is $O(s + \lg N)$. The minimum interesting value for s is $\lg N$, where we achieve space $O((R+C)\lg N + N)$ bits and $O(\lg N)$ time for the operations. We can reduce the $O(N)$ extra space to $o(N)$ by increasing s, which impacts query times and makes them superlogarithmic.

The scheme can be extended to sequences $S[1,N]$ over a small alphabet $[1,\rho]$. The only difference is that the nonterminals x must store $r_c(x)$ for each $c \in [1,\rho]$. Similarly we must store ρ rank values at each sampled position. This raises the overall space to $O(R\rho \lg N + C \lg R + (N\rho/s)\lg N)$. The time stays the same.

In practice. There are several ways to represent $\mathcal{R}$ in compressed form. We choose one [9] that allows for random access to the rules. It represents $\mathcal{R}$ in the form of a directed acyclic graph (DAG) as a sequence S_R and a bitmap S_B. A node is identified as a position in S_B, where a 1 denotes and internal node and a 0 a leaf. The two children of $S_B[i] = 1$ are written next to i, thus we obtain all the subtree by traversing $S_B[i\ldots]$ until we have seen more 0s than 1s. The 0s in S_B are associated to positions in S_R (that is, $S_B[i] = 0$ is associated to $S_R[rank_0(S_B,i)]$). Those leaf symbols are either terminals or nonterminals. Nonterminals are actually positions in S_B that must be recursively expanded. This DAG representation takes, in good cases, as little as 50% of the space required by a plain array representation of $\mathcal{R}$ [9].

In order to reduce the $O(R \lg n)$ space required to store $\ell(x)$ and $r(x)$ for nonterminals x, we store the data only for some of them and obtain the others via expanding the nonterminals. Given a parameter δ, we guarantee that no nonterminal in $\mathcal{C}$ requires expanding at depth more than δ to determine its length and number of 1s. That is, we expand each $\mathcal{C}[i]$ until depth δ or until reaching an already sampled nonterminal. Those nonterminals at depth δ are then sampled. We set up a bitmap B_δ where each sampled nonterminal has a 1, and store $\ell(x)$ and $r(x)$ of marked nonterminal x at an array $E[rank_1(B_\delta, x)]$.

We use a RePair implementation by ourselves (available at `www.dcc.uchile.cl/gnavarro/software`). It has a variant that, although does not guarantee balancedness, has always produced a grammar of very small height in our experiments. The variant that ensures balancedness harms compression in practice.

5 Grammar Compression of Wavelet Trees

Given now a sequence $S[1, N]$ over alphabet $[1, n]$, we build the wavelet tree of S and represent its bitmaps using the compressed format of Section 4. Consider a RePair representation $(\mathcal{R}, \mathcal{C})$ of S, where the sizes of the components is R and C as before. Now take the top-level bitmap B of the wavelet tree. Bitmap B can be regarded as the result of mapping the alphabet of S onto two symbols, 0 and 1. Thus, a grammar $(\mathcal{R}', \mathcal{C}')$ where the terminals are mapped accordingly, generates B. Since the number of rules in $\mathcal{R}'$ is still R and that of $\mathcal{C}'$ is C, the representation of B requires $O(R \lg N + C \lg R + (N/s) \lg N)$ bits (this is of course pessimistic; many more repetitions could arise due to the mapping).

The bitmaps stored at the left and right children of the root correspond to a partition of S into two subsequences S_1 and S_2. Given the grammar that represents S, we can obtain the one that represents S_1 and S_2 by removing all the terminals in the right sides that do not belong to the proper subalphabet, and removing rules with right hands of length 0 or 1. Thus at worst the left and right side bitmaps can also be represented within $O(R \lg N + C \lg R)$ bits each, plus $O((N/s) \lg N)$ for the whole level. Added over the n wavelet tree nodes, the overall space is no more than n times that of the RePair compression of S. The time for the operations raises to $O((s + \lg N) \lg n)$.

Although this does not look alphabet-friendy, and actually the upper bounds are no better than applying the method of Section 4 on a large alphabet ($\rho = n$), the analysis is a (very) pessimistic upper bound. Still one can expect that the repetitions exploited by RePair get cut by half as we descend one level of the wavelet tree, so that after descending some levels no repetition structure can be identified and RePair compression becomes ineffective.

In practice. As n is large, we use a wavelet tree design that concatenates all the bitmaps of the same wavelet tree level [2]. We use one set of rules $\mathcal{R}$ per level.

As the repetitions that could be present in S get shorter when we move deeper in the wavelet tree, we evaluate at each level whether our RePair-based compression is actually better than an entropy-compressed representation [18] or even a plain one, and choose the one with smallest space. Moreover, as *rank* and *select* operations are significantly slower on our RePair-compressed representation, we use a parameter $0 < \alpha \leq 1$ so that we prefer RePair compression only when its size is α times that of the alternatives, or less.

6 Experimental Results

In this section we compare various practical alternatives to document listing and top-k document retrieval. We have chosen four collections of different nature, such as English, Chinese, biological, and symbolic sequences. We show the bpc of its global CSA divided by $\lg \sigma$ to give an idea of its compressibility ratio.

ClueChin: A 2.3 MB sample of ClueWeb09 (`http://boston.lti.cs.cmu.edu/Data/clueweb09`), formed by 23 Web pages in Chinese. Ratio: 5.34/7.99=0.68.

ClueWiki: A 141 MB sample of ClueWeb09, formed by 3,334 Web pages from the English Wikipedia. Ratio: 4.74/6.98=0.68.

KGS: A 75 MB collection of 18,838 sgf-formatted Go game records from year 2009 (`http://www.u-go.net/gamerecords`). Ratio: 4.48/6.93=0.65.

Proteins: A 60 MB collection formed by 143,244 sequences of Human and Mouse proteins (`http://www.ebi.ac.uk/swissprot`). Ratio: 6.02/6.57= 0.92.

Our tests were run on a Intel Core2 Duo machine, 3Ghz, with 8GB RAM and 6MB cache. Our code was compiled using g++ with full optimization.

As the CSA search for the interval $[sp, ep]$ corresponding to a pattern p is common to all the approaches, we do not consider the time for this search (which never exceeds 0.02 milliseconds per query) nor the space for that global CSA (shown for each collection in the previous itemization), but only the extra space/time to support document retrieval once $[sp, ep]$ has been determined. We give the space usage in bits per text character (bpc), and measure user times.

Sadakane's representation [20] builds a separate CSA for each document. For this sake we use a very competitive variant [13,2] available at *PizzaChili* (`http://pizzachili.dcc.uchile.cl/indexes/SSA`). It uses a plain and fast representation for bitmap B (from `http://libcds.recoded.cl`), and an efficient implementation for the two RMQs (from `http://www.uni-ulm.de/in/theo /research/sdsl`). For the space we charge only $2N$ bits for each RMQ structure and zero for B, to account for possible future space reductions.

Our grammar compressed wavelet trees offer a space/time tradeoff depending on the α value, which can be the same for all levels, or decreasing for the deeper levels (where one visits more nodes and thus being slower has a higher impact). Another space/time tradeoff is obtained with the sampling parameter s. We only show one alternative with $\alpha = 1$ and one best-performing alternative with $\alpha < 1$.

As explained, alternative solutions [20,11] for the basic document listing problem are hardly improvable. They require very little extra space and are likely to perform similarly to wavelet trees in time. Our experiments focus on document listing with frequencies, and in top-k retrieval.

6.1 Document Listing with Frequencies

Previous work [3] has demonstrated that the quantile approach [8] is clearly preferable, in space and time, over previous ones based on wavelet trees [22]. Therefore we carry out the quantile algorithm over a plain wavelet tree representation (*WT-Plain*), over one where the bitmaps are statistically compressed [18] (*WT-RRR*), and over our RePair-compressed ones. As explained, we show a variant with $\alpha = 1$ (*WT-RP*, which at each level chooses the lowest space between RePair, plain, or statistically compressed bitmap), and the best performing policy we tried for choosing $\alpha < 1$ values (*WT-RP alpha*).

We also compare Sadakane's approach [20] (*SADA*) on collection *ClueChin*. The construction times over the other collections, with many more documents, was extremely high. This tiny collection will be sufficient to expose the practicality problems of this approach.

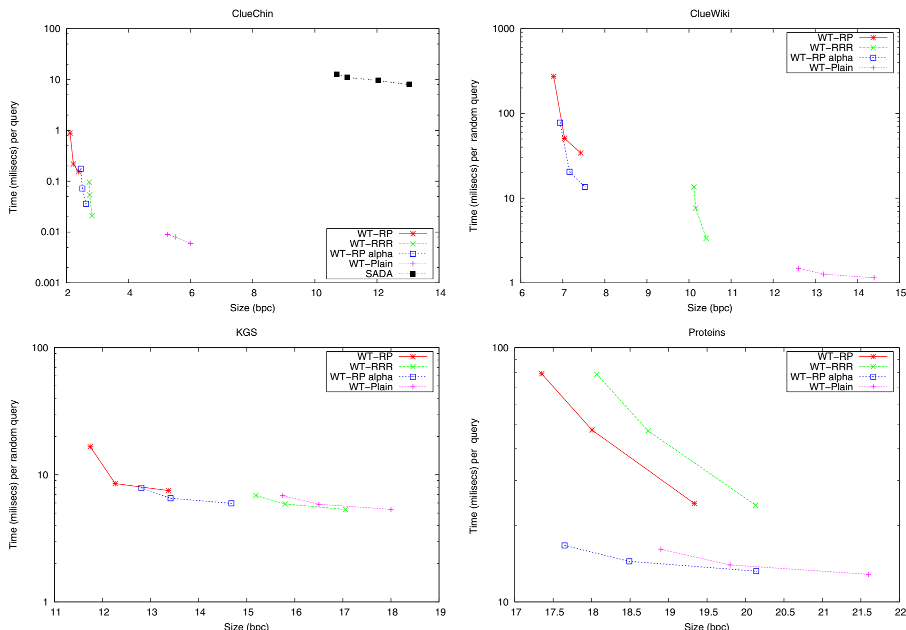


Fig. 1. Experiments for document listing with term frequencies

We chose 10,000 random intervals of the form $[sp, ep]$, considering values for $ep - sp$ from $1,000$ to $10,000$, and listed the distinct documents in the interval with their frequencies. The relative positions of the curves were the same for every $ep-sp$ value, so for lack of space we show just the plots for $ep-sp = 10,000$.

Fig. 1 shows the results. Even on *ClueChin*, with just 23 documents, the space overhead of indexing them separately makes Sadakane's approach impractical (even with the generous assumptions on extra bitmaps and RMQs). It is also slower. For this reason we do not compare Hon et al.'s variant [11], that achieves $|\mathsf{CSA}| + o(n)$ extra space, because it will be much slower and the reduction on space (by $4N$ bits), would be insufficient to make it competitive.

The results are different depending on the type of collections, but in general our compressed representation is able to reduce the space of the plain wavelet tree by a wide margin. The compressed size is 40% to 75% of the original size. The exception is *Proteins*, where the text is mostly incompressible and this translates into the incompressibility of the document array.

While the *WT-RP* is significantly slower than *WT-Plain* (up to 20 times slower in the most extreme case), the *WT-Alpha* versions provide useful tradeoffs. They achieve compression ratios of 50% to 80% and significantly reduce time gaps, to 7 times slower in the most extreme case. The answer time over the interval $[sp, ep]$ of length 10,000 is around 10-20 milliseconds (msecs). We note that our slowest version is still 10 times faster than *SADA*. It is also much faster than listing all the documents individually (e.g., 500 times faster on *ClueChin*).

6.2 Top-k Retrieval

Culpepper et al. [3] present and test two heuristics for top-k retrieval, which run on wavelet trees. The one called *greedy* is always superior, so we test that one in this paper, over our different wavelet tree representations.

We also compare Hon et al.'s structure [11]. Short of implementing it, we (quite) optimistically simulate its performance by charging zero time and zero space to some parts of the data structure and search process. We combine it only with the most promising wavelet tree in each plot.

Recall that the method divides the suffix array A into blocks of fixed length g. After finding the suffix array interval $A[sp, ep]$ corresponding to a pattern, the part of the search corresponding to the blocks fully contained in $[sp, ep]$ is solved with a sampled suffix tree (which we have not implemented and will assume costs zero space and time). The other two subintervals from sp to the start of the next block, and from the end of the last block to ep, are solved by brute force, that is, extracting all the distinct documents and computing their frequency in the whole interval $A[sp, ep]$. This part will be actually executed in different ways. Finally, the candidates obtained by brute force and those given by the suffix tree are ranked and merged (which we will not do and will assume costs zero).

Various alternatives to extract the values of D and compute their frequency are considered. *WT-RP-HON* and *WT-RP-alpha HON* use the corresponding wavelet tree variants for this task. In case the interval $[sp, ep]$ contains no blocks, they switch to Culpepper et al.'s method. On *ClueChin* we tried other variants related to Sadakane's solution [20]. *SADA-HON* uses Sadakane's structure just as in the document listing experiment. *HON* does not use the two RMQ structures, but instead maps the start of the interval $[sp, ep]$ to the local CSA using A and A^{-1}, and then binary searches the end of the local interval by mapping each probe back to the global CSA [11]. Finally, *Search-HON* simply searches for p in the local CSA in order to determine its frequency.

Fig. 2 shows the results. We selected 1,000 substrings at random positions, of length 3 and 6, and retrieved the top-k documents for each, for $k = 1$ and 10. Longer patterns produce shorter $[sp, ep]$ intervals. The relative space and time performance comparisons are similar to those of document listing with frequencies. Most times are around a few tens of msecs per query.

With respect to Hon et al.'s method, *Search-HON* and *HON* are very similar in time, much slower and $4N$ bits smaller than *SADA-HON*. Yet, none of those variants of the original formulation [11] is competitive in practice. What is much more interesting is their combination with a wavelet tree. While it is slightly inferior to Culpepper et al.'s *greedy* heuristic on collections *ClueChin* and *ClueWiki*, on the other two Hon et al.'s method speeds up the heuristic by a factor of up to 1.5–6.5 for $k = 1$ and 2.2–2.5 for $k = 10$. While this is a space- and time-optimistic simulation of Hon et al.'s method, it should be quite tight.

Final remarks. We have shown that the wavelet tree is the best data structure to compute frequencies and support top-k algorithms, and reduced its size by up to 40% while answering within tens of msecs. Also, theoretical top-k proposals

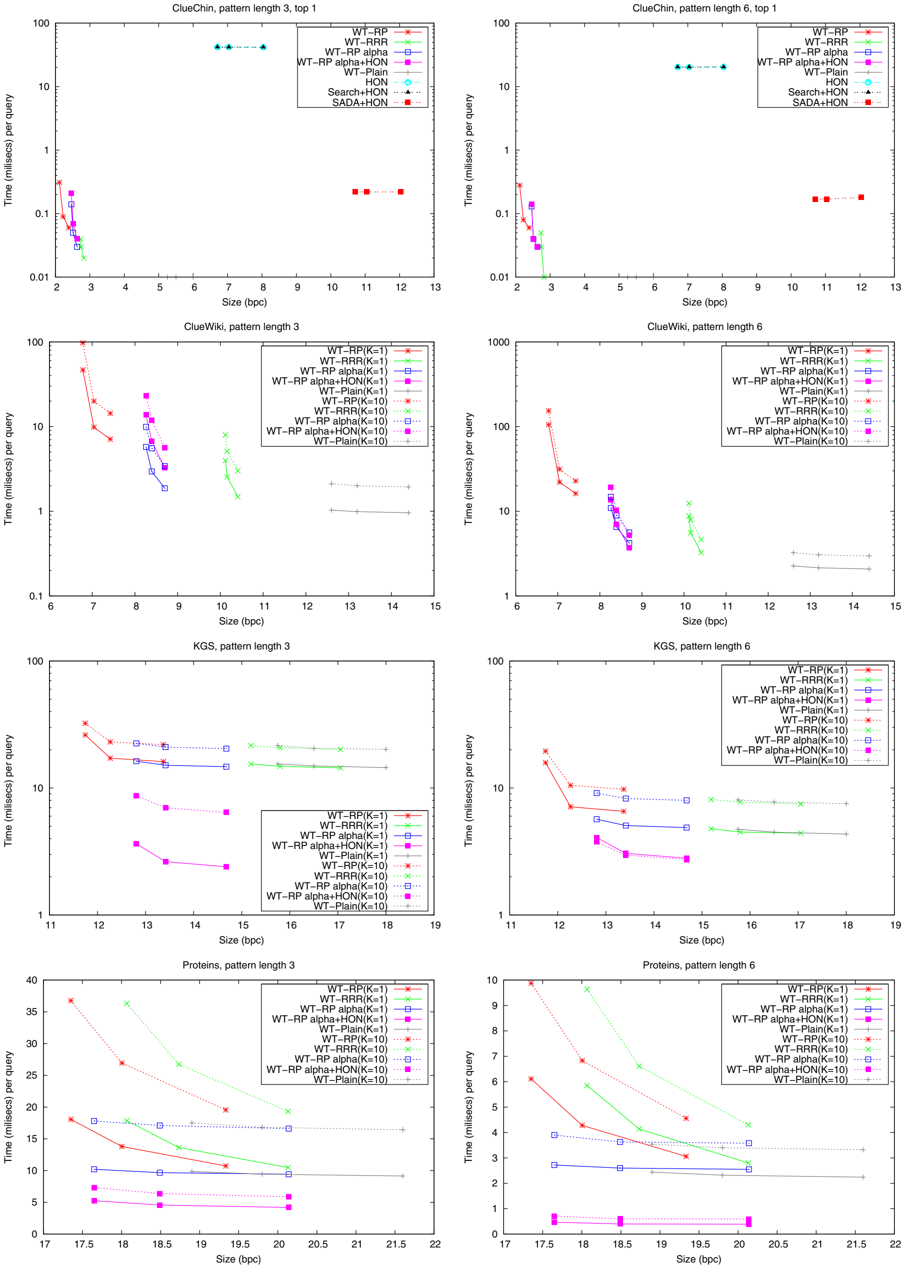


Fig. 2. Experiments for top-k queries

[11] are shown to be worth implementing. Yet, even our smaller indexes (except on the tiny *ClueChin*) are even bigger than the CSAs of the collections (7-17 vs 4.5-6.0 bpc), thus there is much room for improvement in document retrieval. We are still far from the asymptotic space optimality achieved for pattern matching.

References

1. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. NATO ISI Series, pp. 85–96. Springer, Heidelberg (1985)
2. Claude, F., Navarro, G.: Practical rank/Select queries over arbitrary sequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 176–187. Springer, Heidelberg (2008)
3. Culpepper, S., Navarro, G., Puglisi, S., Turpin, A.: Top-k ranked document search in general text databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
4. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM JEA 13, article 12 (2009)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), article 20 (2007)
6. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 459–470. Springer, Heidelberg (2007)
7. Gagie, T., Navarro, G., Puglisi, S.J.: Colored range queries and document retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)
8. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
9. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 216–227. Springer, Heidelberg (2007)
10. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850 (2003)
11. Hon, W.-K., Shah, R., Vitter, J.: Space-efficient framework for top-k string retrieval problems. In: FOCS, pp. 713–722 (2009)
12. Larsson, N.J., Moffat, J.A.: Offline Dictionary-Based Compression. Proc. of the IEEE 88, 1722–1732 (2000)
13. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 229–241. Springer, Heidelberg (2007)
14. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comp. 22(5), 935–948 (1993)
15. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
16. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: SODA, pp. 657–666 (2002)
17. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)

18. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA, pp. 233–242 (2002)
19. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Alg. 48(2), 294–313 (2003)
20. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. Discr. Alg. 5(1), 12–22 (2007)
21. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. J. Discr. Alg. 3, 416–430 (2005)
22. Välimäki, N., Mäkinen, V.: Space-efficient algorithms for document retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)

Influence of Pruning Devices on the Solution of Molecular Distance Geometry Problems

Antonio Mucherino[1], Carlile Lavor[2], Therese Malliavin[3], Leo Liberti[4], Michael Nilges[3], and Nelson Maculan[5]

[1] CERFACS, Toulouse, France
mucherino@cerfacs.fr
[2] IMECC, UNICAMP, Campinas, SP, Brazil
clavor@ime.unicamp.br
[3] Institut Pasteur, Paris, France
{terez,michael.nilges}@pasteur.fr
[4] LIX, École Polytechnique, Palaiseau, France
liberti@lix.polytechnique.fr
[5] COPPE, Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil
maculan@cos.ufrj.br

Abstract. The Molecular Distance Geometry Problem (MDGP) is the problem of finding the conformation of a molecule from inter-atomic distances. In some recent work, we proposed the *interval* Branch & Prune (iBP) algorithm for solving instances of the MDGP related to protein backbones. This algorithm is based on an artificial ordering given to the atoms of the protein backbones which allows the discretization of the problem, and hence the applicability of the iBP algorithm. This algorithm explores a discrete search domain having the structure of a tree and prunes its infeasible branches by employing suitable pruning devices. In this work, we use information derived from Nuclear Magnetic Resonance (NMR) to conceive and add new pruning devices to the iBP algorithm, and we study their influence on the performances of the algorithm.

1 Introduction

Proteins are important molecules formed by chains of smaller molecules called amino acids. Several experimental techniques, as Nuclear Magnetic Resonance (NMR), are able to provide some information on interatomic distances in protein molecules which can be exploited for obtaining the three-dimensional conformation of the protein. As the protein conformation often enables to give good clues about the protein function, the conformation determination is of fundamental importance. The problem of finding the protein conformation from a list of inter-atomic distances is known in the scientific literature as the Molecular Distance Geometry Problem (MDGP) [4]. By nature, the MDGP is a constraint satisfaction problem, but its solution is usually attempted by employing global optimization techniques [10]. It usually requires a search in a continuous space which is a subset of $\mathbb{R}^{3n}$, where n is the number of atoms forming the molecule. It has been proved that the MDGP is an NP-hard problem [16].

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 206–217, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Since 2006 we have been working on a combinatorial reformulation of the MDGP. Under suitable assumptions, we are able to discretize the problem and to reduce the search on a discrete search domain. Even though the problem is still NP-hard after the discretization [3], it can be efficiently solved by employing a Branch & Prune (BP) algorithm [9]. It is important to remark that this algorithm is able to find *all* solutions to the problem, differently from other algorithms based on continuous formulations and/or heuristics [10,15].

We refer to this combinatorial reformulation of the MDGP as Discretizable MDGP (DMDGP) [3]. Let $G = (V, E, d)$ be a weighted undirected graph representing an instance of the problem: each vertex in V corresponds to an atom, and there is an edge in E between two vertices if and only if the distance between the corresponding atoms is known (the distance value is given by the associated weight d). In order to have the combinatorial reformulation, we need two assumptions to be satisfied for a given ordering on the vertices in V. By Assumption 1, the edge set E must contain all cliques on quadruplets of consecutive vertices, that is,

$$\forall i \in \{4, \ldots, n\}\ \forall j, k \in \{i-3, \ldots, i\} \qquad (\{j, k\} \in E)$$

and, by Assumption 2, the following strict triangular inequality

$$\forall i = 2, \ldots, n-1, \quad d_{i-1,i+1} < d_{i-1,i} + d_{i,i+1},$$

must hold.

Assumption 1 ensures that the distances between each possible pair of atoms in any quadruplet of consecutive atoms are known. Moreover, if Assumption 2 holds, there cannot be triplets of consecutive atoms that are perfectly aligned. Supposing that positions for the atoms are searched by following the same ordering given to the vertices of V, there exist at most two possible positions in which each atom can be placed if these two assumptions are satisfied. This leads to the definition of a discrete search domain, which has the structure of a tree. This tree can be constructed in the practice by exploiting distances that must be known by Assumption 1. Moreover, the considered instance can also contain other distances, that we can use for pruning branches of the tree in order to focus our searches on its feasible branches only. This is the main idea behind the BP algorithm [9].

The basic version of this algorithm has however two main limitations. First of all, exact distances should be available in order to construct the discrete search domain, whereas real-life NMR data are usually noisy, so that lower and upper bounds on the distances are actually known. Moreover, given any atom of the protein, there must be at least 3 distances concerning this atom, otherwise Assumption 1 cannot be satisfied. This property is quite difficult to be satisfied by NMR instances, because the number of available distances is usually not sufficient, and only distances related to particular atoms, mainly pairs of hydrogens, are actually available. Therefore, even though the BP algorithm is extremely efficient in its basic version, it is unfortunately mainly suitable for simulated instances of the DMDGP, and not for NMR instances.

We recently overcame these two issues by introducing a hand-craft ordering for the atoms of the protein backbones, and by proposing an extension of the BP algorithm which is based on such an ordering. This ordering allows us to discretize a full class of MDGPs, the one which is related to protein backbones, even if only noisy distances between pairs of hydrogens are available. This is possible because all distances used in the construction of the discrete search domain can be computed a priori by information on the chemical composition of the protein backbones. The distances obtained by NMR are only used for pruning purposes. We shall refer to this extension of the BP algorithm as *interval* BP (*i*BP) [6].

The *i*BP algorithm is able to consider NMR instances related to protein backbones. However, in our previous publications [5,6], we only presented computational experiments where simulated data were considered. Indeed, when we firstly tried to solve NMR instances by *i*BP, we found out that the available information on the distances was not sufficient for efficiently pruning the search domain. For this reason, we decided to conceive new pruning devices, with the aim of identifying sooner during the search infeasible parts of the search domain. These new pruning devices are all based on other information (rather than distances) that NMR experiments can provide. We also analyze the influence of each newly added pruning device on the performances of the *i*BP algorithm.

The rest of the paper is organized as follows. In Section 2, we give a brief description of the *i*BP algorithm and of the artificial ordering for the protein backbones which allows the discretization of the problem. New pruning devices are presented in details in Section 3, and computational experiments on NMR instances are given in Section 4. Conclusions are drawn in Section 5.

2 The Interval Branch and Prune

In order to solve DMDGPs where interval data are considered, we recently defined an artificial ordering for the atoms of the protein backbones. In this section, we describe this particular artificial ordering and we discuss the *i*BP algorithm, that is based on this ordering. For more details, the interested reader is referred to [5,6].

Let us start by assigning the following ordering to the atoms of the first amino acid of the considered protein:

$$r^1_{PB} = \{N^1, H^1, H^0, C^1_\alpha, N^1, H^1_\alpha, C^1_\alpha, C^1\}.$$

Note that the superscripts indicate the amino acid to which each atom belongs. One of the hydrogens bound to N^1 (in general, there is only one hydrogen) is indicated by the symbol H^0. The carbon C^1_α and the nitrogen N^1 appear twice in the sequence. This is done in order to reduce the relative distances between pairs of atoms in the ordering, and also in order to consider the distances between copies of the same atom (that must be equal to 0). The other carbon of the first amino acid, the atom C^1, is considered, in this case, only once. Let us now assign the following ordering to the atoms of the second amino acid:

$$r^2_{PB} = \{N^2, C^2_\alpha, H^2, N^2, C^2_\alpha, H^2_\alpha, C^2, C^2_\alpha\}.$$

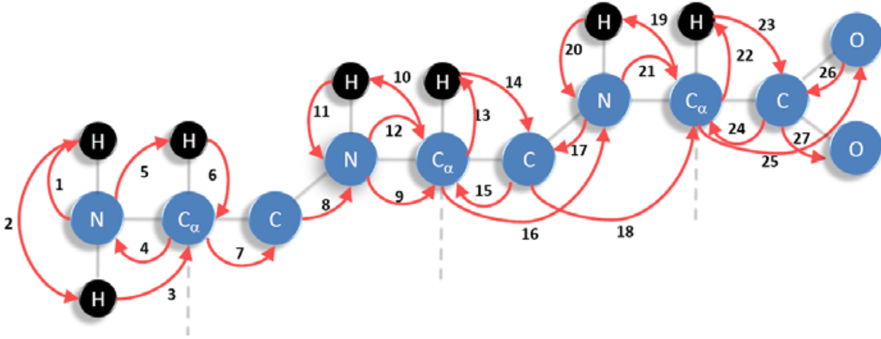


Fig. 1. The hand-craft artificial ordering r_{PB}

This sequence of atoms is used for building a *bridge* between the first amino acid, and the third one, from which a generic ordering is considered. In fact, the ordering defined on the second amino acid is quite similar to the generic one. Atoms are considered more than once, and, in particular, the carbon C_α^2 appears in the sequence 3 times. This is the ordering for the generic amino acid (from the third to last but one):

$$r_{PB}^i = \{N^i, C^{i-1}, C_\alpha^i, H^i, N^i, C_\alpha^i, H_\alpha^i, C^i, C_\alpha^i\}.$$

The nitrogen N^i is considered twice, the carbon C_α^i is considered 3 times, and the carbon C^{i-1} belonging to the previous amino acid is repeated among the atoms of the amino acid i. In total, for each amino acid, we have 4 copies of atoms that already appeared somewhere else in the sequence. Note that hydrogen atoms are never duplicated. Since the last amino acid contains a few atoms more, this is the ordering that we consider:

$$r_{PB}^p = \{N^p, C^{p-1}, C_\alpha^p, H^p, N^p, C_\alpha^p, H_\alpha, C^p, C_\alpha^p, O^p, C^p, O^{p+1}\}.$$

Note that this is the only case in which oxygen atoms appear. The two oxygens O^p and O^{p+1} present in the last residue r_{PB}^p correspond to the two oxygens of the C-terminal carboxyl group COO^- of the protein.

Let us indicate by the symbol r_{PB} the defined artificial ordering on the whole protein backbone:

$$r_{PB} = \{r_{PB}^1, r_{PB}^2, \ldots, r_{PB}^i, \ldots, r_{PB}^p\}.$$

Fig. 1 shows the hand-craft ordering for a small protein backbone formed by 3 amino acids. It is constructed so that, for each atom $v \in V$, the three edges $(v-3, v)$, $(v-2, v)$ and $(v-1, v)$ are always contained in E. The corresponding distances are obtained from known bond lengths and bond angles, that only depend from the kind of bound atoms. The two edges $(v-2, v)$ and $(v-1, v)$ are always associated to exact distances, whereas only the edge $(v-3, v)$ may be associated to an interval distance. In particular, there are three different

Algorithm 1. The iBP algorithm

```
1: iBP(j, r, d, D)
2: if (r_j is a duplicated atom) then
3:    iBP(j + 1, r, d, D);
4: else
5:    if (d(r_j − 3, r_j) is exact) then
6:       b = 2;
7:    else
8:       b = 2D;
9:    end if
10:   for k ∈ {1, . . . , b} do
11:      compute the k-th atomic position x^k_{r_j} for the r_j-th atom;
12:      check the feasibility of position x^k_{r_j} using pruning devices;
13:      if (x^k_{r_j} is feasible) then
14:         if (j = |r|) then
15:            a solution x is found, print it;
16:         else
17:            iBP(j + 1, r, d, D);
18:         end if
19:      end if
20:   end for
21: end if
```

possibilities. If $d(v-3, v) = 0$, then v represents a duplicated atom, and therefore the only feasible coordinates for v are the same of its previous copy. If $d(v-3, v)$ is an exact distance, the standard discretization process can be applied, and two possible positions for v can be computed. Finally, if $d(v-3, v)$ is represented by an interval, we discretize the interval and take D sample distances from it. For each sample distance, we apply the standard discretization process. In this case, $2 \times D$ possible atomic positions can be computed for v. As a consequence, the discrete search domain is a tree, which is not necessarily binary (this would require that all distances $d(v-3, v)$ are exact) [5].

Algorithm 1 is a sketch of the *interval* BP (iBP) [6]. It essentially requires 4 input arguments: the index j (in the ordering given to V) of the current atom to be placed, the artificial ordering r, the set of distances d (which can be either exact or represented by intervals), and the number D of sample distances used for discretizing interval distances. The main focus of this paper is on line 12 of Algorithm 1: the pruning devices that are used for discovering infeasible atomic positions.

3 Pruning Devices

Pruning devices can be used in the iBP algorithm for pruning away infeasible branches of the discrete search domain. In this work, we study the influence of pruning devices on the performances of the algorithm. Each of such pruning devices is based on a different kind of information which can be obtained

through NMR experiments. The Direct Distance Feasibility (DDF) device (see Section 3.1) considers the available lower and upper bounds on the distances between hydrogen atoms. The Torsion Angle Feasibility (TAF) device (see Section 3.2) considers instead the lower and upper bounds on the protein backbone torsion angles. Finally, the Secondary Structure Feasibility (SSF) device (see Section 3.3) is based on the so-called *chemical shift index* of spin nuclei of the atoms C_α and H_α of each amino acid. Indeed, as shown in [11,18], these indices are strongly related to the secondary structures to which each amino acid belongs. The technique described in [17], for example, is able to compute torsion angle restraints in secondary structures from chemical shift indices, with a precision of about 10°.

3.1 Direct Distance Feasibility (DDF)

NMR experiments are able to provide a list of lower and upper bounds on some distances between pairs of hydrogen atoms of the molecule. The Direct Distance Feasibility (DDF) pruning device is based on the idea of pruning atomic positions for which these lower and upper bounds are not satisfied. DDF has been widely used in our previous publications: even though it represents a very basic test, and it is easy to implement, DDF allows us to discard large parts of the discrete search domain very efficiently on sets of artificial instances [5,7,8,12,14]. However, when we tried to consider real NMR data, we noticed that the range defined by these lower and upper bounds is so large that DDF is not able anymore to sufficiently prune branches of the tree. This causes the multiplication of the solutions found by iBP, where some infeasible solutions are also contained. This is the reason why we needed to add new pruning devices in order to consider NMR instances.

3.2 Torsion Angle Feasibility (TAF)

Along with the list of lower and upper bounds on the distances, NMR experiments can also provide information on the torsion angles of protein backbones. Three different torsion angles can be defined along the backbone main chain $N - C_\alpha - C - N - \ldots$:

$$\begin{aligned} \phi &\equiv \{C, N, C_\alpha, C\}, \\ \psi &\equiv \{N, C_\alpha, C, N\}, \\ \omega &\equiv \{C_\alpha, C, N, C_\alpha\}. \end{aligned}$$

The angle ϕ, for example, is the angle defined by the two planes $\{C, N, C_\alpha\}$ and $\{N, C_\alpha, C\}$. The torsion angle ω is usually very close to π, because there is a peptide bond that does not allow this subset of atoms to take any other configuration. The other two angles ϕ and ψ, instead, can vary in larger ranges.

Even if the iBP algorithm is not based on the torsion angle representation of the protein backbone, but rather on an atomic representation, the torsion angles ϕ and ψ can be easily computed every time the four atoms needed for their computation are available. As soon as the value for one of these angles is obtained, we can check if it satisfies the known lower and upper bounds provided by NMR: the last positioned atom can be pruned if the computed angle

does not satisfy this constraint. We shall call this pruning device Torsion Angle Feasibility (TAF). Note that it is useless to consider the torsion angle ω because, by construction, our artificial backbone always satisfies the constraint $\omega = \pi$.

3.3 Secondary Structure Feasibility (SSF)

Subsets of atoms of a protein can fold in local structures which are very typical in proteins. Such local structures are referred to as *secondary structures*, and they are mainly represented by α-helices and β-sheets. In both cases, these secondary structures are stabilized through hydrogen bonds between pairs of amino acids. More precisely, given a pair (a_i, a_j) of amino acids belonging to the same secondary structure, there is a hydrogen bond between the hydrogen H (the one bound to N) of amino acid a_i and the oxygen O (bound to C) of amino acid a_j. This hydrogen bond forces the involved atoms, and in particular the hydrogen H of a_i and the oxygen O of a_j, to be very close to each other.

As a consequence, the torsion angles ϕ and ψ are constrained to vary in predefined ranges when the corresponding amino acids fold in α-helix or β-sheet. The bounds on the torsion angles can therefore be refined by using this information. Moreover, in the case of α-helices, it is known that the amino acid a_j is always a_{i+4}. Therefore, the two atoms which need to be closer than a certain threshold are known a priori: a new distance (between the hydrogen H of a_i and the oxygen O of a_{i+4}) can be added to the list of known distances. The possibility to add this new distance for each amino acid in α-helices reflects the strong regularity of this secondary structure; β-sheets are instead less regular: for each a_i, we do not know a priori the corresponding a_j.

In order to reject conformations which do not satisfy the restrictions given by the protein secondary structures, we use the *chemical shift index* described above to predict the subset of amino acids that are supposed to fold in α-helix or in β-sheet. As mentioned above, the technique described in [17] is able to find good estimates of the torsion angles related to amino acids having a given chemical shift index. However, since we do not need in general tight bounds on the torsion angles, we just consider intervals that are centered in $-60°$ for both ϕ and ψ (typical values for α-helices), or centered in $135°$ and $-120°$ (typical values for ϕ and ψ, respectively, in β-sheets) [1].

The Secondary Structure Feasibility (SSF) pruning device is therefore based on the idea of refining bounds for the torsion angles and/or adding new distances to the considered instance. This is done by exploiting information obtained by NMR on the chemical shift index of each amino acid. When the secondary structure is an α-helix, the oxygen O bound to the carbon C is needed for verifying the hydrogen bond distance. Note that this oxygen is not included in our artificial ordering (see Fig. 1). However, we can easily compute its coordinates when the positions for the atom C (which is bound to O), for the atom N (which is bound to C) and for the atom H (which is bound to N) are known. Because of the presence of a peptide bond (the same which forces the torsion angle ω to be equal to π), the four atoms O, C, N and H lie on the same plane. Bond lengths are known, and, since bond angles are also known, the distance between O and

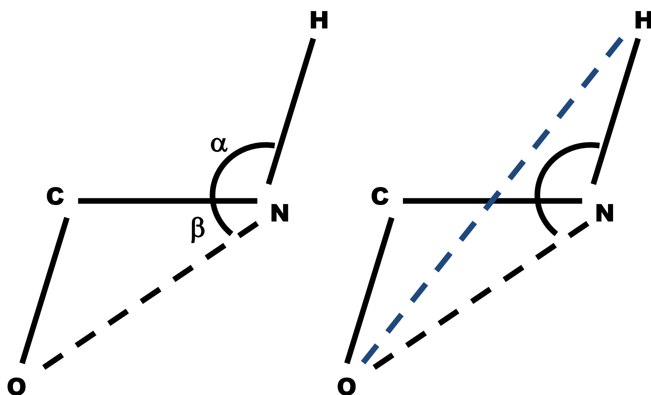


Fig. 2. Distances and angles that allow to compute the distance between the oxygen O bound to C and the hydrogen H bound to N. The angle β can be computed from known information regarding the triangle $\widehat{OCN}$. α is instead a bond angle. The distance between O and H can be computed by solving the triangle $\widehat{ONH}$, where the angle in N is $\alpha + \beta$. Note that the four atoms must lie on the same plane. The same procedure can also be applied for the other possible configuration, when the torsion angle is 0.

N, as well as the distance between C and H, can be computed. Finally, Fig. 2 shows how to compute the distance between O and H. By exploiting all these distances and the coordinates for the atoms C, N and H, the coordinates for O can be uniquely computed.

The coordinates of the oxygen O can be computed by intersecting three spheres which are centered in the three atoms C, N and H, and having as radii the corresponding distances from O. This intersection of spheres can be computed by solving two linear systems: this same idea is also exploited in a generalization of the DMDGP presented in [13], and the interested reader can find many details about this procedure in the reference paper. We just remark that the procedure can provide in general two possible sets of coordinates for the oxygen O. Because of numerical errors, this may happen even if only one position for O is actually feasible. In our implementation, the pruning device SSF is applied only if the computed coordinates of O are not affected by numerical errors.

4 Computational Experiments

The iBP algorithm has been implemented in C programming language and compiled by the GNU C compiler v.4.1.2 with the `-O3` flag. We performed the experiments presented in this section on an Intel Core 2 CPU 6400 @ 2.13 GHz with 4GB RAM, running Linux.

For the first time since we started to work on this topic, we are able to present computational experiments where real data from NMR are managed by using an algorithm based on a discrete search. The data related to protein conformations having different features (number of amino acids, secondary structures)

Table 1. Experiments on NMR instances. #DDF, #TAF and #SSF provide the number of times each pruning device found and discarded an infeasible atomic position. The symbol "–" indicates that the corresponding pruning device was not used in the experiment.

instance name	n_{aa}	n	D	iBP calls	#DDF	#TAF	#SSF	CPU time
2jmy	15	134	15	4724652	2356670	-	-	39
2jmy	15	134	15	10482	5244	2695	-	1
2jmy	15	134	15	31986247	15206046	-	6189223	248
2jmy	15	134	15	33709275	16017742	1069321	5156934	298
2ppz	36	323	20	98807	48586	-	-	1
2ppz	36	323	20	91466	43568	41600	-	2
2ppz	36	323	20	414926692	142727215	-	70158539	10263
2ppz	36	323	20	58296108	18941155	10111249	615926	1471
2jwu	56	503	22	6528633	6715391	-	-	117
2jwu	56	503	22	11159985	28183553	1029437	-	396
2jwu	56	503	22	20119294	14742376	-	1601915	432
2jwu	56	503	22	44795676	19494850	9450743	5313997	1363

have been downloaded from the Protein Data Bank (PDB) [2], along with the corresponding conformations already obtained by other methods. For each instance, we study the influence of the implemented pruning devices on the iBP algorithm. We point out that, in general, NMR instances may not contain the necessary information for applying all three pruning devices. Distances are always available, and therefore the pruning device DDF can always be considered. Bounds on torsion angles and chemical shift indices might be omitted. In the following, we consider instances where all necessary information is supposed to be available.

Table 1 shows the results of our experiments for a selected subset of instances. The instance name is the PDB code of the molecule. `2jmy` is a small peptide completely folded in α-helix. `2ppz` is a protein containing only α-helices as secondary structure, whereas `2jwu` contains both secondary structures. All considered instances contain information on distances and torsion angles. Only distances regarding the hydrogens H and H_α of each amino acid have been considered: all others (mainly related to the amino acid side chains) have been discarded. The information regarding the secondary structures have been obtained from the conformations downloaded from the PDB (we plan to include a procedure to automatically interpreting the chemical shift index associated to each amino acid in future versions of iBP, see for example [17]). In the table, n_{aa} is the number of amino acids forming the protein, whereas n is the length of the artificial ordering r_{PB} (hence, it consists of the number of atoms in the protein backbone, including duplicated atoms). The number D of sample distances which are considered for discretizing intervals is also specified for each experiment. The behavior of the iBP algorithm is evaluated through the number of times the algorithm recursively calls itself before finding the first solution, and through the number of times each pruning device is able to identify and prune an infeasible atomic

position. If this information is absent (the symbol "–" is used in the table), it means that the pruning device was not applied in the given experiment. For each protein, we performed 4 experiments, where the following combinations of pruning devices were considered: DDF, DDF+TAF, DDF+SSF, DDF+TAF+SSF. The *i*BP algorithm is stopped as soon as the first solution is found. For each experiment, we provide the CPU time in seconds.

We consider the number of *i*BP calls as a valid measure of the influence of the newly added pruning devices. When the number of *i*BP calls decreases, the added pruning devices were able to discard infeasible atomic positions that DDF was not able to recognize, and lead the search towards feasible positions sooner. In this case, the CPU time decreases. Moreover, when the number of *i*BP calls instead increases, atomic positions previously considered as feasible are declared infeasible by the new pruning devices, and therefore the search is focused on different parts of the search domain. In this case, the CPU time may increase, but there is a gain on the quality of the obtained solution.

Both situations can be seen in Table 1. For the helix `2jmy`, the number of *i*BP calls decreases of two orders of magnitude when the pruning device TAF is added to the standard DDF. Indeed, #DDF decreases, because TAF was able to recognize infeasible atomic positions earlier during the search and was able to prune larger parts of the search domain. This also happens when TAF is added to DDF alone or DDF+SSF in the experiments related to the protein `2ppz`. Otherwise, the second situation is more common in these experiments: when a new pruning device is added, the number of atomic positions pruned by these devices while working in cooperation increases. Therefore, they are able to lead the search towards better solutions, i.e. towards solutions where the constraints related to all pruning devices are satisfied.

We remark that only one solution to the problem is computed in these experiments. At this stage of our work, we cannot analyze yet the influence of the pruning devices on the whole set of solutions, because the considered pruning devices, even if they are used all together, are not able to keep under control the combinatorial explosion due to the recursive calls to *i*BP. For the same reason, we cannot judge yet on bio-related aspects of the found solutions in comparison to the employed pruning devices. We plan to do so in the future by including the amino acid side chains in our artificial backbone.

5 Conclusions

The *i*BP algorithm for the MDGP is the first algorithm implementing a discrete search which is able to manage interval data. It can be currently applied to MDGPs related to protein backbones, for which we identified a particular artificial ordering for their atoms that allows us to discretize the problem. In this work, we studied the influence of pruning devices on a set of NMR instances, i.e. instances where real data from NMR are contained. The pruning devices are based on different information that can be obtained through NMR experiments: a list of bounds on the distances between pairs of atoms of the molecule, a list of bounds on the torsion angles of the protein backbones, and finally information

regarding the protein secondary structures. The presented experiments showed that the newly added pruning devices are actually able to prune away large parts of the discrete search domain, so that the search can be focused on feasible parts of the domain. Next step is to consider information regarding the amino acid side chains. This could allow us to identify only a few feasible solutions for each considered instance.

Acknowledgments

The authors wish to thank the Brazilian research agencies FAPESP and CNPq, the French research agency CNRS, Institut Pasteur, and École Polytechnique, for financial support.

References

1. Berg, J.M., Tymoczko, J.L., Stryer, L.: Biochemistry, 6th edn. W.H. Freeman publications, New York (2006)
2. Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T.N., Weissig, H., Shindyalov, I.N., Bourne, P.E.: The protein data bank. Nucleic Acid Research 28, 235–242 (2000)
3. Lavor, C., Liberti, L., Maculan, N.: The discretizable molecular distance geometry problem. Technical Report q-bio/0608012, arXiv (2006)
4. Lavor, C., Liberti, L., Maculan, N.: Molecular distance geometry problem. In: Floudas, C., Pardalos, P. (eds.) Encyclopedia of Optimization, 2nd edn., pp. 2305–2311. Springer, New York (2009)
5. Lavor, C., Liberti, L., Mucherino, A.: On the solution of molecular distance geometry problems with interval data. In: IEEE Conference Proceedings, International Workshop on Computational Proteomics, International Conference on Bioinformatics & Biomedicine (BIBM 2010), Hong Kong, pp. 77–82 (2010)
6. Lavor, C., Liberti, L., Mucherino, A.: The *i*BP algorithm for the discretizable molecular distance geometry problem with interval data (submitted) (Available on Optimization Online)
7. Lavor, C., Mucherino, A., Liberti, L., Maculan, N.: On the computation of protein backbones by using artificial backbones of hydrogens. To appear in Journal of Global Optimization (2011); Available online from July 24, 2010
8. Lavor, C., Mucherino, A., Liberti, L., Maculan, N.: Discrete approaches for solving molecular distance geometry problems using NMR data. International Journal of Computational Biosciences 1(1), 88–94 (2010)
9. Liberti, L., Lavor, C., Maculan, N.: A branch-and-prune algorithm for the molecular distance geometry problem. International Transactions in Operational Research 15, 1–17 (2008)
10. Liberti, L., Lavor, C., Mucherino, A., Maculan, N.: Molecular distance geometry methods: from continuous to discrete. International Transactions in Operational Research 18(1), 33–51 (2010)
11. Mielke, S.P., Krishnan, V.V.: An evaluation of chemical shift index-based secondary structure determination in proteins: influence of random coil chemical shifts. Journal of Biomolecular NMR 30(2), 143–196 (2004)

12. Mucherino, A., Lavor, C.: The branch and prune algorithm for the molecular distance geometry problem with inexact distances. In: Proceedings of the International Conference on Computational Biology, vol. 58, pp. 349–353. World Academy of Science, Engineering and Technology (2009)
13. Mucherino, A., Lavor, C., Liberti, L.: The Discretizable Distance Geometry Problem (submitted)
14. Mucherino, A., Liberti, L., Lavor, C., Maculan, N.: Comparisons between an exact and a metaheuristic algorithm for the molecular distance geometry problem. In: Rothlauf, F. (ed.) Proceedings of the Genetic and Evolutionary Computation Conference, Montreal, pp. 333–340. ACM, New York (2009)
15. Nilges, M., Gronenborn, A.M., Brunger, A.T., Clore, G.M.: Determination of three-dimensional structures of proteins by simulated annealing with interproton distance restraints. application to crambin, potato carboxypeptidase inhibitor and barley serine proteinase inhibitor 2. Protein Engineering 2, 27–38 (1988)
16. Saxe, J.B.: Embeddability of weighted graphs in k-space is strongly NP-hard. In: Proceedings of 17th Allerton Conference in Communications, Control and Computing, pp. 480–489 (1979)
17. Shen, Y., Delaglio, F., Cornilescu, G., Bax, A.: TALOS+: a hybrid method for predicting protein backbone torsion angles from NMR chemical shifts. Journal of Biomolecular NMR 44(4), 213–236 (2009)
18. Wishart, D.S., Sykes, B.D., Richards, F.M.: The chemical shift index: a fast and simple method for the assignment of protein secondary structure through NMR spectroscopy. Biochemistry 31(6), 1647–1698 (1992)

An Experimental Evaluation of Treewidth at Most Four Reductions

Alexander Hein[1] and Arie M.C.A. Koster[2]

[1] Student of Mathematics, RWTH Aachen University
[2] Lehrstuhl II für Mathematik, RWTH Aachen University,
Wüllnerstr. 5b, D-52062 Aachen, Germany
koster@math2.rwth-aachen.de

Abstract. We analyze the computational effectiveness of the graph reductions proposed by Sanders [12,13] to recognize graphs of treewidth at most four. We show that graphs of treewidth at most four can be recognized extremely fast by this infinite set of reductions. For graphs of larger treewidth, however, the added value of the specific reductions for treewidth four fades away with the width.

1 Introduction

Graphs of bounded *treewidth* are of particular importance for the development of polynomial time algorithms for, in general, NP-complete combinatorial optimization problems. Hence, there exists a theoretical and practical interest in determining the treewidth of a graph (cf. [5,6]) as well as characterizing graphs of low treewidth (cf. [2]), although computing the treewidth of a given graph is an NP-complete problem [1].

Connected graphs of treewidth one are exactly trees, whereas graphs of treewidth two correspond to series-parallel graphs [8]. If k is a constant, linear time algorithms exist to check whether or not the treewidth is at most k. Bodlaender's algorithm [3] was experimentally evaluated in [11] and it was concluded to be computationally intractable for k as small as four.

Graphs of treewidth at most four can be recognized by sets of reductions. Graphs of treewidth at most three can be reduced to the null graph by six reductions derived in [2]. These reductions can also be used to preprocess graphs of larger treewidth as was, along with further preprocessing rules, shown and experimentally evaluated in [7]. In [12,13], a linear time algorithm to recognize graphs of treewidth at most four is presented, consisting of an infinite set of reductions. In contrast to the reductions for treewidth at most three, no experimental evaluation has been performed yet.

The purpose of this paper is twofold. On the one hand, we report on the first implementation of the linear time algorithm to recognize graphs of treewidth at most four. We show that this algorithm is computationally tractable and evaluate the frequencies the reductions apply. On the other hand, we show that the reductions for treewidth at most four provide an additional preprocessing possibility for graphs of larger treewidth.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 218–229, 2011.
© Springer-Verlag Berlin Heidelberg 2011

This paper is organized as follows. In Section 2 all preliminaries are discussed, starting with the definition of treewidth, and ending with the linear time algorithm for treewidth at most four. Next, in Section 3 the reductions are analyzed for randomly generated graphs of treewidth at most four, whereas in Section 4 their effectiviness for preprocessing graphs of larger treewidth is studied. The paper is closed with some concluding remarks. This paper is based on [10].

2 Preliminaries

Let $G = (V, E)$ be an undirected graph consisting of a set of vertices V and a set of edges E. By $N_G(v)$ we denote the set of neighbors of v in graph G. K_n and W_n denote resp. a complete graph and a wheel on n vertices.

Several equivalent notions for treewidth have been studied over time, in particular *tree decompositions*, *partial k trees* and *elimination orderings*. Here, we restrict to the latter two. A graph is a k-tree if recursively removing a vertex with a clique of k vertices as neighbors results in a K_k. A *partial-k-tree* is a subgraph of a k-tree. The smallest k for which G is a partial-k-tree is the *treewidth* $\tau(G)$ of G. An *elimination ordering* of a graph $G = (V, E)$ is a bijection $f : V \to \{1, 2, \ldots, n\}$. An elimination ordering f is *perfect*, if for all $v \in V$, the set of its higher numbered neighbors $\{w \mid \{v, w\} \in E \ \wedge \ f(w) > f(v)\}$ forms a clique. A graph G has treewidth at most k if and only if there exists a supergraph $H \supseteq G$ having a perfect elimination ordering f and at most k higher numbered neighbors for every vertex. In this case, f is called a *k-elimination sequence* of G.

A reduction is formally described by a replacement of a specific subgraph by another (smaller) subgraph: A *structure* S is a pair $(G(S), (u_1, \ldots, u_j))$ where G is a graph and $u_1, \ldots, u_j$ are distinct vertices of G, called the *vertices of attachment of* S or short *anchors*. A graph *has* a structure S if $G(S)$ is a subgraph of G and $N_{G(S)}(v) = N_G(v)$ for all non-anchors $v \in G(S)$ (i.e., only non-anchors might be connected to the rest of G).

A *reduction* R is a pair of structures, S_R and T_R, with the same anchors and $|V(S_R)| > |V(T_R)|$. For graphs G, H and reduction R, the graph G is reduced to H by R if G has S_R, H has T_R and H is obtained from G by replacing S_R by T_R. For each reduction R the partial order $H \leq_R G$ implies that there is a sequence of graphs such that $H = G_1, G_2, \ldots, G_k = G$ such that for $0 < i \leq k$ G_i is reduced to G_{i-1} by R.

A reduction R is *TW_k-safe* if for all graphs G and H with $H \leq_R G$, $\tau(G) \leq k$ if and only if $\tau(H) \leq k$. A set of reductions Q is *TW_k-complete* if for all graphs G $\tau(G) \leq k$ if and only if G can be reduced to the null graph by reductions in Q. Note that R is TW_{k+1}-safe if R is TW_k-safe.

2.1 Reductions for Graphs of Treewidth at Most Three

In [2] a set of six reductions for recognizing graphs of treewidth at most three have been derived, see Table 1 and Figure 1 where black vertices are anchors. Reduction *zero* (or *islet*) is TW_0-safe, reduction *one* (or *twig*) is TW_1-safe and

Table 1. Description of the reductions for treewidth at most three

R	structure S_R	structure T_R
zero	let $v \in V$ of degree zero	remove v
one	let $v \in V$ of degree one	remove v
series	let $v \in V$ with $N_G(v) = \{u, w\}$	add uw to E; remove v
triangle	let $v \in V$ with $N_G(v) = \{x, y, z\}$ such that $xy \in E$	add xz and yz to E; remove v
buddy	let $v, w \in V$ with $N_G(v) = N_G(w) = \{x, y, z\}$	add xy, xz, yz to E; remove v, w
cube	let $d \in V$ with neighbors $\{a, b, c\}$ only connected with $\{u, v, w\}$ as in Figure 1	add uv, uw, vw to E; remove a, b, c, d

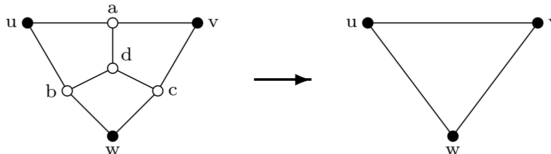


Fig. 1. Reduction *cube*

together TW_1-complete. Reduction *series* is TW_2-safe and together with *zero* and *one* TW_2-complete. The reductions *triangle*, *buddy*, and *cube* are TW_3-safe and all six together are TW_3-complete.

2.2 Reductions for Graphs of Treewidth at Most Four

In addition to the reductions for treewidth at most three, Sanders [12,13] derived an (infinite) set of reductions for treewidth at most four. Reductions YO, $H7$, TO, YI, $L1$, $L2$, $L3$, and $L4$ are described best by Figure 2. In the four so-called *Y-Δ reductions* $Y0$, $H7$, TO, and YI, a vertex of degree three (a Y) is replaced by a triangle on its neigbhors (a Δ). Also in the so-called *ladder reductions* $L1$ to $L4$, one (or two) vertices of degree three are replaced by (a) triangle(s).

All above reductions are not sufficient to characterize graphs of treewidth at most four. To obtain a complete set of reductions, Sanders [12,13] introduced 60 simple *leaf structures* (structures possible at the leaves of a tree decomposition) and four *infinite families of leaf structures*. Figure 3 shows the central leaf structures (all other structures are slight extensions; a complete list can be found in [10,13]), whereas Figure 4 shows the families of leaf structures, consisting of a repetition of the middle part with four different startings and two different endings. Structures S_{buddy} and S_{cube} can be seen as the union of two resp. three LS_1 leaf structures (where anchors are identified).

The remaining reductions are defined by particular unions of leaf structures: A *superstructure* S is a structure $(G(S), (u_1, \ldots, u_j))$ with $j \leq 4$ and a *center* vertex $x \notin A := \{u_1, \ldots, u_j\}$. The graph $G(S)$ is the union of the graphs of a finite set L of leaf structures, such that for each $M \in L$, the set B_M of anchors satisfies $x \in B_M \subset A \cup \{x\}$. S_{cube} is a superstructure with d as center vertex.

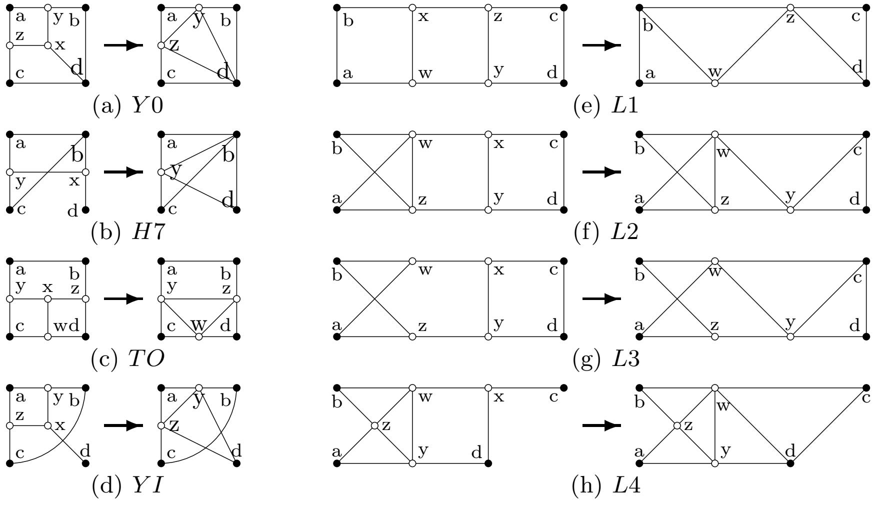


Fig. 2. Reductions YO, H7, TO, YI, L1, L2, L3, and L4

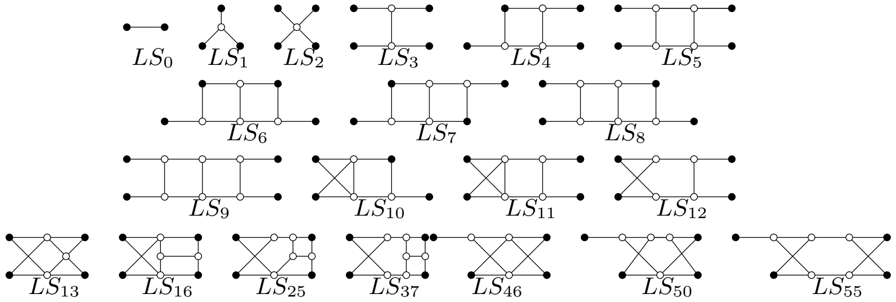


Fig. 3. Central leaf structures

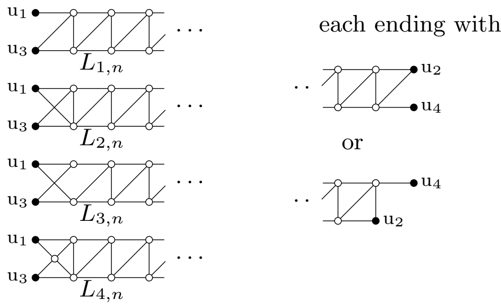


Fig. 4. Infinite classes of leaf structures

Hence, *cube* is not necessary in the set of TW_4-complete reductions. S_{buddy}, however, does not have a center.

To define the TW_4-complete set of reductions, Sanders [12,13] introduced the *Star-O* (SO) reduction: $S_{SO} = (W_5, (a, b, c, d))$ and $T_{SO} = (K_4, (a, b, c, d))$. The set CM (clique-minor) consists of all reductions R with S_R a superstructure, T_R a complete graph on the anchors of S_R, and either $G(T_R)$ is a minor of $G(S_R)$ or there is a minor J of G such that $H \leq_{SO} J$. Sanders proves that the set $CS_4 := \{zero, one, series, triangle, \ buddy, YO, H7, TO, YI, L1, L2, L3, L4\} \cup CM$ is TW_4-complete. Note that this set has an infinite number of members.

2.3 Linear Time Algorithm for Graphs of Treewidth at Most Four

To obtain a linear time algorithm to recognize graphs of treewidth at most four, Sanders [12,13] had to guarantee that the center vertex of a CM-reduction has degree at most a constant, e.g., 20. Therefore, an alternative set of TW_4-safe reductions, *Triple*, is defined by all reductions R with $T_R = (K_4, (a, b, c, d))$ and $G(S_R)$ is the union of three leaf structures having a subset of $\{a, b, c, d\}$ as anchors, none of which is LS_0, and no two of which are LS_1 with the same set of anchors. Finally, BCM is the set of reductions R with $R \in CM$ but S_R does not contain buddy or a reduction in *Triple*. By this modification, the center of each $R \in BCM$ has degree at most 20 and together with *Triple* we still have a TW_4-complete set of reductions.

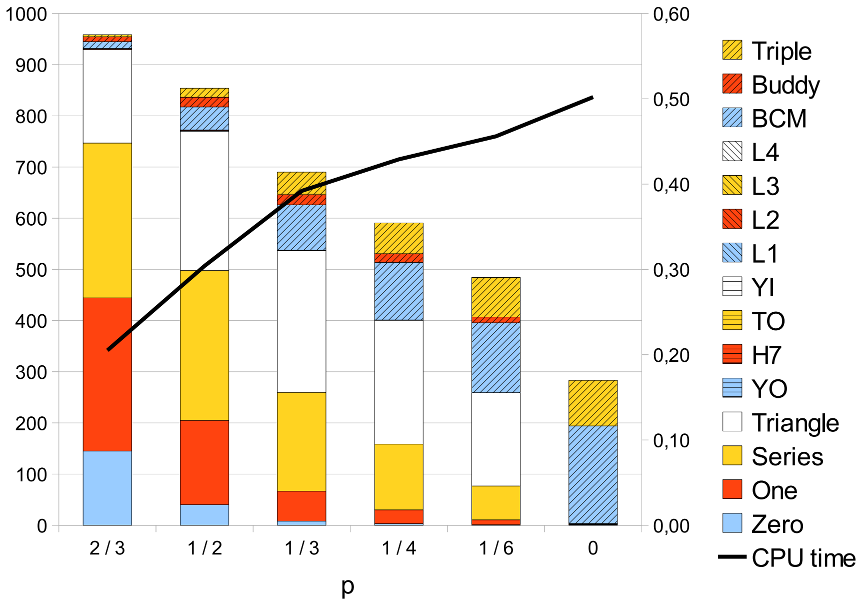
Theorem 1 (Sanders [12, 13]). *Given a graph G, there exists a linear time algorithm that determines a 4-elimination sequence (i.e., $\tau(G) \leq 4$) or returns a reduced graph H with $\tau(G) = \tau(H) \geq 5$.*

Space limitations prohibit a detailed description of the algorithm. Sanders designed a subroutine to check whether a vertex is the center of a CM reduction. It either determines that the vertex cannot be the center of a CM reduction or finds a *buddy*, *BCM*, or *Triple*. Since the center check for each vertex requires already linear time, this subroutine can only take constant time. The key here is that the center has a degree of at most 20.

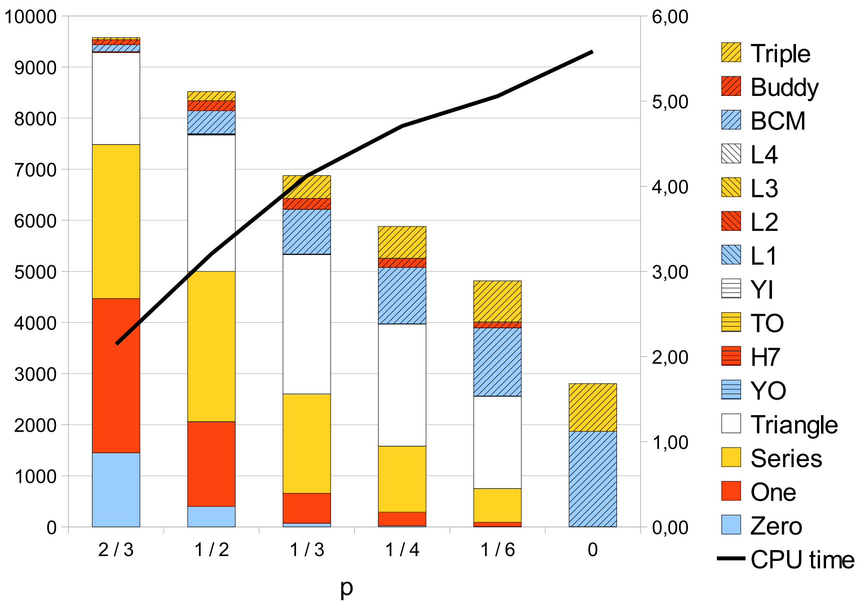
Sanders shows that a vertex can be contained in at most four leaf structures as internal vertex. The main routine keeps track of this. It works with a stack of vertices to be checked. Each time a reduction is applied, the information on leaf structures must be updated for the anchors only and these anchors have to be added to the stack. Since each reduction has at most four anchors, the stack is empty after at most $5n$ pops. After a pop, first, the *easy* reductions from Table 1 and Figure 2 are checked. If no reduction is found, the subroutine is called. For further details on the algorithm, we refer to [12,13,10].

3 Evaluation of Treewidth at Most Four Reductions

The linear time algorithm for recognizing graphs of treewidth at most four has been implemented in C++ and tested for randomly generated partial-4-trees. The partial-4-trees have been generated by randomly removing a percentage p



(a) $n = 1,000$, $\#G = 10,000$



(b) $n = 10,000$, $\#G = 1,000$

Fig. 5. Average success rate of the reductions (first Y-axis) and CPU time in seconds (second Y-axis) for partial-4-trees with p of the edges removed at random

Table 2. Total number of applications of reductions, average number of vertices reduced per reduction, and average CPU time in seconds for different settings of number of vertices n, randomly edges removed p and number of generated graphs $\#G$

n	1,000	1,000	1,000	1,000	1,000	1,000	10,000	10,000	10,000	10,000	10,000	10,000
p	2/3	1/2	1/3	1/4	1/6	0	2/3	1/2	1/3	1/4	1/6	0
$\#G$	10,000	10,000	10,000	10,000	10,000	10,000	1,000	1,000	1,000	1,000	1,000	1,000
zero	1,449,989	406,043	80,952	30,634	14,124	10,000	1,449,953	401,238	73,137	21,985	5,310	1,000
one	2,993,031	1,644,768	585,364	270,991	92,294	10,000	3,018,187	1,658,682	584,233	266,147	84,762	1,000
series	3,025,430	2,928,603	1,931,273	1,283,895	660,954	10,000	3,016,243	2,940,219	1,944,771	1,291,375	661,851	1,000
triangle	1,828,458	2,722,820	2,764,959	2,421,235	1,826,061	4,460	1,796,951	2,672,875	2,726,930	2,390,180	1,807,885	415
YO	490	746	327	159	33	0	561	716	367	136	28	0
$H7$	17,454	18,827	9,591	4,770	1,473	0	17,298	18,486	9,529	4,758	1,472	0
TO	46	36	5	0	0	0	46	41	5	3	0	0
YI	2,193	1,502	428	144	29	0	2,084	1,510	501	126	25	0
$L1$	3	2	0	0	0	0	5	0	0	0	0	0
$L2$	8	6	1	0	0	0	7	5	2	2	0	0
$L3$	18	16	4	1	0	0	26	15	3	0	1	0
$L4$	83	114	54	13	2	0	68	95	47	25	11	0
BCM	132,015	450,549	889,021	1,124,081	1,361,685	1,906,605	136,603	451,420	876,139	1,104,347	1,334,225	1,867,047
buddy	95,241	189,228	206,055	170,670	112,214	0	96,418	195,216	212,407	177,890	116,079	0
Triple	40,626	176,458	431,895	599,996	772,231	889,687	40,764	180,663	448,396	623,220	803,976	933,413
Total	9,585,085	8,539,718	6,899,929	5,906,589	4,841,100	2,830,752	9,575,214	8,521,181	6,876,467	5,880,194	4,815,625	2,803,875
#vertices/R	1.0	1.2	1.4	1.7	2.1	3.5	1.0	1.2	1.5	1.7	2.1	3.6
CPU time (s)	0.2	0.3	0.4	0.4	0.5	0.5	2.1	3.2	4.1	4.7	5.1	5.6

of the edges in a 4-tree. Considered values for p are 2/3, 1/2, 1/3, 1/4, 1/6, and (for reference) 0. A total of 10,000 resp. 1,000 random partial-4-trees with $n = 1,000$ resp. $n = 10,000$ have been generated. The computations have been carried out on a notebook with a 2.5 GHz CPU and 4 Gb RAM. Detailed results are listed in Table 2 and summarized in Figure 5.

First of all note that, for $p = 0$ the reductions *zero*, *one*, and *series* are applied exactly once per graph. These are the last three reductions before the null graph is obtained. For larger values of p the frequencies increase (rapidly) as the partial-4-trees become more and more loosely connected.

The opposite behaviour is true for the *BCM* and *Triple* reductions. These are more frequently applied for dense graphs. As those reductions result in a removal of several vertices, the total number of reductions decreases if the density increases. Reductions *triangle* and *buddy* seem to play an important role in all cases (except for $p = 0$ where *buddy* cannot be applied by construction). All other reductions are hardly ever applied.

The line *#vertices/R* in Table 2 reports the average number of vertices removed per reduction. Note that *zero*, *one*, and *series* remove a single vertex. Hence, with increasing p the number of vertices per reduction decreases.

Finally, the computation times show that (i) the reductions can be applied very effectively to recognize graphs of treewidth four, (ii) the time increases with the density of the graph, and (iii) the time is indeed linear with the size of the graph. Note that, Röhrig [11] has reported computational intractability of the linear time algorithm of Bodlaender [3] as soon as $k \geq 4$ (an actual comparison of running times was not possible as both the original author and we were not able to rerun the implementation of [11]).

4 Preprocessing of General Graphs

The described reductions can also be applied to graphs of treewidth larger than four. In such cases, the graph will be reduced to a non-null graph with same treewidth. In this section, we study the added value of the TW_4-safe reductions, compared to the TW_3-safe reductions for graphs of larger treewidth.

First, we generated, similar to Section 3, partial-k-trees for $k = 5$, $k = 7$, and $k = 10$. For each k and p a total of 10,000 randomly generated partial-k-trees with $n = 1,000$ vertices have been generated. Figure 6 and Table 3 show the average success rate of the reductions for various p values. The results show that for $k = 5$ and a high number p of removed edges, the TW_3-safe reductions significantly reduce the size of the graph. The success rate of *BCM* can still be observed, whereas *buddy* and *Triple* are rather rarely successful. For $k = 7$ and $k = 10$ the rates decrease rapidly, where high p values cause loosely connected parts that are preprocessed by *zero*, *one*, *series*, and *triangle*. The probability that a *BCM* (*buddy*, *Triple*) reduction can be applied is fading away as k increases. Like for $k = 4$, the other reductions (YO–$L4$) are even less frequently applied. Since the majority of reductions remove only one vertex, the average number of vertices removed per reduction is close to one. The line *#vertices left* shows the average size of the remaining graph. Computation times are again very small.

Table 3. Average numbers of reductions, average size of graph after reduction, and CPU time in seconds for 10,000 graphs per setting of partial-k-tree, the number of vertices n, randomly edges removed p

k	5	5	5	5	5	7	7	7	7	7	10	10	10	10	10
n	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
p	2/3	1/2	1/3	1/4	1/6	2/3	1/2	1/3	1/4	1/6	2/3	1/2	1/3	1/4	1/6
zero	93.1	19.4	2.4	0.5	0.1	40.4	4.6	0.3	0.0	0.0	11.8	0.5	0.0	0.0	0.0
one	259.6	104.2	24.2	8.2	1.7	152.1	35.3	3.7	0.6	0.1	61.9	5.9	0.2	0.0	0.0
series	289.7	226.4	102.4	51.5	17.7	246.8	111.7	22.9	6.7	1.0	146.8	29.2	1.8	0.2	0.0
triangle	153.1	164.3	107.0	67.8	31.9	98.8	64.2	16.6	4.8	0.7	45.7	9.9	0.3	0.0	0.0
YO	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
$H7$	2.1	1.5	0.4	0.1	0.0	1.4	0.3	0.0	0.0	0.0	0.3	0.0	0.0	0.0	0.0
TO	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
YI	0.3	0.1	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
$L1$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
$L2$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
$L3$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
$L4$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
BCM	18.8	26.5	20.7	13.8	6.5	3.4	1.8	0.3	0.1	0.0	0.3	0.0	0.0	0.0	0.0
buddy	4.9	3.2	0.8	0.2	0.0	0.8	0.1	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0
Triple	4.1	3.8	1.5	0.6	0.1	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Total	825.8	549.5	259.4	142.8	58.1	543.9	218.1	43.8	12.2	1.7	266.9	45.6	2.3	0.2	0.0
#vertices left	120.6	382.4	693.7	827.8	928.7	448.3	778.6	955.7	987.7	998.2	732.6	954.4	997.7	999.8	1,000.0
#vertices/R	1.1	1.1	1.2	1.2	1.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
CPU time (s)	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.2	0.2	0.1	0.3	0.2	0.1	0.1	0.1

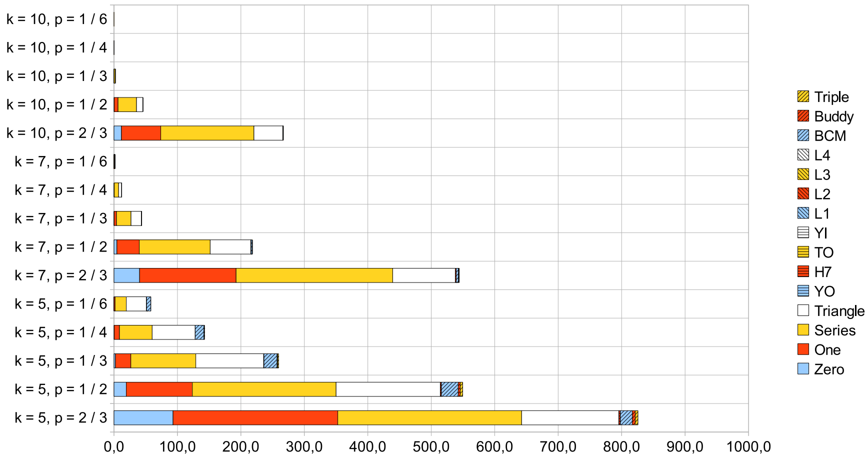


Fig. 6. Average success rates of the reductions for partial-k-trees with p of the edges removed randomly (10,000 randomly generated partial-k-trees with $n = 1,000$)

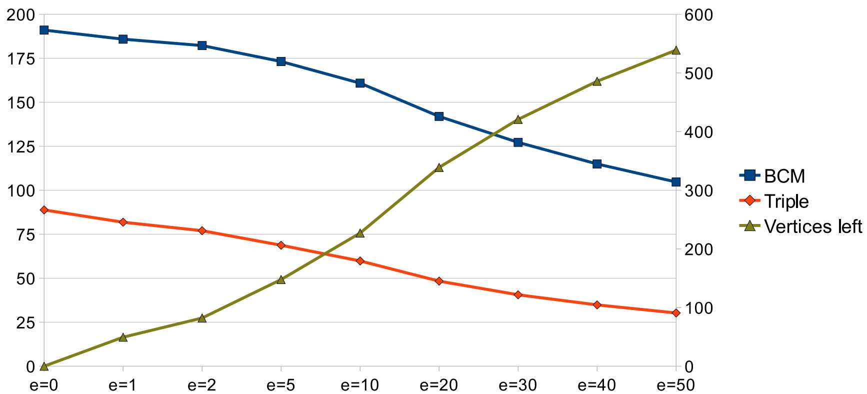


Fig. 7. Average success rates (first y-axis) and remaining graph size (second y-axis) for 4-trees with e new edges (1,000 randomly generated graphs with $n = 1,000$)

Alternatively, one can added edges to a 4-tree. By adding a single edge, the treewidth will increase (but not more than one). We randomly added $e = 0, 1, 2, 5, 10, 20, 30, 40, 50$ edges to randomly generated 4-trees with $n = 1000$. Except for $e = 0$, only *BCM* and *Triple* were applied successfully. Figure 7 shows the average number of reductions by these rules and the average remaining number of vertices. As expected, the number of reductions decreases if e increases. Already for a single edge, graphs of about 50 vertices remain after reduction (note that the graphs have 3991 edges in total). With 50 extra edges, more than half of the vertices cannot be removed anymore.

Table 4. Preprocessing by TW_k-safe and reductions for selected probabilistic networks (1st block), frequency assignment (2nd block), coloring (3rd block), and TSP (4th block) graphs (all other rules did not have any effect)

	original		TW_3		TW_4								
instance	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	*zero*	*one*	*series*	*triangle*	*BCM*	$\|V\|$	$\|E\|$	reduced	time (s)
alarm	37	65	5	10	1	7	15	12	1	0	0	100.0%	0.00
diabetes	413	819	212	492	1	2	100	146	47	0	0	100.0%	0.24
fungiuk	15	36	10	22	1	2	3	4	2	0	0	100.0%	0.00
mildew	35	80	12	27	1	1	4	23	2	0	0	100.0%	0.00
pignet2	3,032	7,264	1,051	3,835	0	51	1,322	608	1	1,049	3,830	65.4%	0.60
celar06	100	350	74	313	0	7	7	12	2	69	299	31.0%	0.01
celar07	200	817	153	747	0	12	26	20	3	134	704	33.0%	0.02
celar08	458	1,655	302	1,427	9	29	55	76	14	251	1,295	45.2%	0.06
anna	138	986	68	363	0	28	22	20	1	65	354	52.9%	0.01
david	87	812	63	361	0	11	5	8	0	63	361	27.6%	0.01
homer	561	3,258	177	1,125	11	237	86	53	3	168	1,100	70.1%	0.06
huck	74	602	53	269	2	9	5	5	1	51	262	31.1%	0.00
d198	198	571	189	552	0	0	0	9	1	185	542	6.6%	0.02
d493	493	1,467	488	1,452	0	0	0	5	0	488	1,452	1.0%	0.06
d657	657	1,958	650	1,938	0	0	0	7	1	647	1,930	1.5%	0.09
d1291	1,291	3,845	1,290	3,843	0	0	0	1	0	1,290	3,843	0.1%	0.16
rl1304	1,304	3,879	1,286	3,826	0	0	0	18	20	1,225	3,663	6.1%	0.21
rl1323	1,323	3,950	1,306	3,900	0	0	0	17	18	1,252	3,754	5.4%	0.20
rl1889	1,889	5,631	1,851	5,519	0	0	0	38	62	1,665	5,017	11.9%	0.31
rl5915	5,915	17,728	5,842	17,510	0	0	0	73	106	5,523	16,642	6.6%	0.88
rl5934	5,934	17,770	5,866	17,571	0	0	0	68	99	5,569	16,770	6.2%	0.90

In earlier preprocessing studies [7,9] the TW_3-safe reductions have been applied to particular graphs of applications like TSP, coloring, frequency assignment, and probabilistic networks. Typically, such graphs behave different from randomly generated graphs. To avoid a distortion of the results, we have, in contrast to [7], not applied generalizations of the six TW_3-safe reductions like *simplicial* and *almost simplicial* to reduce graphs of treewidth at least four further. In Table 4 a (biased) selection of those graphs is listed. For each instance, we report $|V|$ and $|E|$ in the original and after application of the TW_k-safe reductions for $k = 3, 4$. In addition, for $k = 4$, the number of applied reductions is reported as well as the percentage of vertices removed, and CPU time.

Without exception, the reductions $YO, H7, TO, YI, L1, L2, L3$, and $L4$ as well as *buddy* and *Triple* cannot be applied. Occassionaly, *BCM* could be applied, in particular for frequency assignment graphs and probabilistic networks. Four of the five selected probabilistic networks have treewidth four and are reduced to the null graph. Most impressive, by the new reductions 212 vertices of *diabetes* are removed. Except for TSP graphs, the reductions *one*, *series*, and *triangle* are most effective. The TSP graphs are Delaunay triangulations of TSP instances and therefore do not have vertices of low degree. The reductions *triangle* and, less frequently, *BCM* reduce the number of vertices in the selected TSP graphs between 0.1% and 11.9%.

5 Conclusions

In this paper, we reported on a computational evaluation of the linear time algoirthm to recognize graphs of treewidth at most four by Sanders [12,13].

The algorithm is based on an infinite set of TW_4-safe reductions that reduce G to the null graph if and only if the graph has treewidth at most four. These reductions can also be applied as preprocessing step for graphs of treewidth at least five. Our computational experiments for partial-k-trees and particular application graphs show that the linear time algorithm is remarkable fast to recognize graphs of treewidth at most four. For graphs of treewidth close to four, the TW_4-specific reductions account for an additional, but typically small, reduction of the graph size. The fast majority of the reductions are the well-known TW_3-safe reductions.

Based on these results, a further extension of the theory to a TW_k-complete set of reductions for $k \geq 5$ seems to be not rewarding. Instead, the derivation of further general reductions like *simplicial* and *almost simplicial* and/or safe separators [4] seems to be more promising.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM Journal on Algebraic and Discrete Methods 8, 277–284 (1987)
2. Arnborg, S., Proskurowski, A.: Characterization and recognition of partial 3-trees. SIAM Journal on Algebraic and Discrete Methods 7, 305–314 (1986)
3. Bodlaender, H.L.: A linear time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on Computing 25, 1305–1317 (1996)
4. Bodlaender, H.L., Koster, A.M.C.A.: Safe separators for treewidth. Discrete Mathematics 306, 337–350 (2006)
5. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations I. Upper bounds. Information and Computation 208, 259–275 (2010)
6. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations II. Lower bounds. Technical Report UU-CS-2010-22, Dept. of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands (2010)
7. Bodlaender, H.L., Koster, A.M.C.A., van der Eijkhof, F.: Pre-processing rules for triangulation of probabilistic networks. Computational Intelligence 21(3), 286–305 (2005)
8. Bodlaender, H.L., van Antwerpen-de Fluiter, B.: Parallel algorithms for series parallel graphs and graphs with treewidth two. Algorithmica 29, 543–559 (2001)
9. van der Eijkhof, F., Bodlaender, H.L., Koster, A.M.C.A.: Safe reduction rules for weighted treewidth. Algorithmica 47, 138–158 (2007)
10. Hein, A.: Reduktionsregeln für baumweite 4. Bachelor's thesis, Mathematik, RWTH Aachen University (2010), http://www.math2.rwth-aachen.de/koster
11. Röhrig, H.: Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany (1998)
12. Sanders, D.P.: Linear Algorithms for Graphs of Tree-Width at Most Four. PhD thesis, Georgia Institute of Technology (1993)
13. Sanders, D.P.: On linear recognition of tree-width at most four. SIAM Journal on Discrete Mathematics 9(1), 101–117 (1996)

A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks

Ittai Abraham, Daniel Delling,
Andrew V. Goldberg, and Renato F. Werneck

Microsoft Research Silicon Valley
{ittaia,dadellin,goldberg,renatow}@microsoft.com

Abstract. Abraham et al. [SODA 2010] have recently presented a theoretical analysis of several practical point-to-point shortest path algorithms based on modeling road networks as graphs with low highway dimension. They also analyze a *labeling algorithm*. While no practical implementation of this algorithm existed, it has the best time bounds. This paper describes an implementation of the labeling algorithm that is faster than any existing method on continental road networks.

1 Introduction

Motivated by computing driving directions, the problem of finding point-to-point shortest paths in road networks has received significant attention in recent years. Even though Dijkstra's algorithm [11] solves it in almost linear time [15], continent-sized road networks require something faster. Preprocessing makes sublinear-time algorithms possible; see [5] for a survey of existing methods.

In particular, goal-directed methods, such as *arc flags* (AF) [16], direct the search towards the target. Hierarchical methods, such as *contraction hierarchies* (CH) [14], sparsify the search space by visiting only important vertices when far from the source or target. *Transit node routing* (TNR) [3,4] reduces long-range queries to a few table lookups, using the fact that on road networks a small set of vertices is enough to hit all long shortest paths out of a region. TNR+AF [5] (combining TNR, CH, and arc flags) is the fastest algorithm for random queries, six orders of magnitude faster than Dijkstra. For local and mid-range queries, CH and High-Performance Multi-Level Routing (HPML) [9] are the fastest.

Although algorithms such as these are known to work well in practice, a theoretical analysis has been given only recently, by Abraham et al. [2]. The method with the best time bounds is a *labeling algorithm*. Labeling algorithms have been studied before in more theoretical settings [6,12,21].

The preprocessing stage of the labeling algorithm computes, for each vertex v, a *forward label* $L_f(v)$ and a *reverse label* $L_r(v)$. Each consists of a set of vertices w, together with their respective distances from (in $L_f(v)$) or to (in $L_r(v)$) v. A labeling is *valid* if it has the *cover property*: for every pair of vertices s and t, $L_f(s) \cap L_r(t)$ contains a vertex u on a shortest path from s to t. An s–t query finds the vertex $u \in L_f(s) \cap L_r(t)$ that minimizes $\text{dist}(s,u) + \text{dist}(u,t)$ and

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 230–241, 2011.
© Springer-Verlag Berlin Heidelberg 2011

returns the corresponding path. Intuitively, a label for v is a set of *hubs* to which v has a direct connection, and any two vertices s and t share at least one hub on the shortest s–t path. Although efficient in theory, the algorithm as described by Abraham et al. [2] is impractical for continent-sized road networks: preprocessing would be too slow, and the worst-case memory usage is prohibitive.

Motivated by theory, we develop HL (Hub-based Labeling algorithm), a practical implementation of the labeling algorithm for road networks. We start from the fact that the sets of vertices visited by the forward and reverse searches of hierarchical algorithms (such as CH) contain the corresponding labels. Similar observations have been made implicitly for graphs of bounded tree-width [12] and road networks [17,14]; we make it explicit and take advantage of it. We then propose several techniques to make our method truly practical. First, we show how to obtain much smaller labels by efficiently pruning the CH search space and applying ideas from the theoretical preprocessing algorithm [2]. Second, we describe how to compress each label. Finally, we show how to implement preprocessing and queries efficiently.

Our main contribution is to show that the labeling algorithm is practical. In fact, our experiments show that HL is currently the fastest algorithm for the problem. When optimized for speed, it answers a random query in as much time as five random accesses to main memory. This is faster than TNR+AF by a factor of more than three, and than HPML by more than an order of magnitude. For local queries, HL is about three times faster than HPML and an order of magnitude faster than TNR+AF. Using compression, we obtain a version of HL with a memory footprint that is comparable to the other two algorithms, but is still faster for all types of queries.

This paper is organized as follows. Section 2 reviews relevant previous work and describes our experimental setup. Section 3 presents the basic version of HL. Section 4 describes several improvements that make it truly practical. Section 5 compares HL with other algorithms experimentally. We conclude in Section 6. The full version of this paper [1] contains details omitted due to space limitations.

2 Preliminaries

The preprocessing stage of a point-to-point shortest path algorithm takes a graph $G = (V, A)$ as input, with $|V| = n$, $|A| = m$, and length $\ell(a) > 0$ for each arc a. The length of a path P in G is the sum of its arc lengths. The query stage takes a source s and a target t as input and returns the distance $\text{dist}(s, t)$ between them.

Dijkstra's algorithm. The standard solution to this problem is Dijkstra's algorithm [11], which processes vertices in increasing order of distance from s. For every vertex v, it maintains the length $d(v)$ of the shortest s–v path found so far, as well as the predecessor $p(v)$ of v on the path. Initially $d(s) = 0$, $d(v) = \infty$ for all other vertices, and $p(v) = \textit{null}$ for all v. At each step, a vertex v with minimum $d(v)$ value is extracted from a priority queue and *scanned*: for each arc $(v, w) \in A$, if $d(v) + \ell(v, w) < d(w)$, we set $d(w) = d(v) + \ell(v, w)$ and $p(v) = w$. The algorithm terminates when the target t is extracted.

Contraction hierarchies. Preprocessing enables much faster exact queries on road networks. The *contraction hierarchies* (CH) algorithm [14], in particular, is based on the notion of *shortcuts* [19]. The *shortcut operation* deletes (temporarily) a vertex v from the graph; then, for any neighbors u, w such that $(u, v) \cdot (v, w)$ is the only shortest path between u and w, it adds a *shortcut arc* (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$, thus preserving the shortest path information.

The CH preprocessing routine defines a total order among the vertices and shortcuts them sequentially in this order, until a single vertex remains. It outputs a graph $G^+ = (V, A \cup A^+)$ (where A^+ is the set of shortcut arcs created), as well as the vertex order itself. We denote the position of a vertex v in the order by $rank(v)$. Define $G^\uparrow = (V, A^\uparrow)$ by $A^\uparrow = \{(v, w) \in A \cup A^+ : rank(v) < rank(w)\}$. Similarly, $A^\downarrow = \{(v, w) \in A \cup A^+ : rank(v) > rank(w)\}$ and $G^\downarrow = (V, A \cup A^\downarrow)$.

During an s–t query, the *forward CH search* runs Dijkstra from s in $G^\uparrow$, and the *reverse CH search* runs reverse Dijkstra from t in $G^\downarrow$. For every $v \in V$, these searches lead to upper bounds $d_s(v)$ and $d_t(v)$ on distances from s to v and from v to t. For some vertices, these estimates may be greater than the actual distances (and even infinite for unvisited vertices). However, as shown by Geisberger et al. [14], the maximum-rank vertex u on the shortest s–t path is guaranteed to be visited, and $v = u$ will minimize $d_s(v) + d_t(v) = \text{dist}(s, t)$.

Queries are correct regardless of the contraction order, but query times and the number of shortcuts added may vary greatly. For best results, on-line heuristics are used to select which vertex to shortcut next [14]. Our implementation [7] sets the priority of a vertex u to $2ED(u) + CN(u) + H(u) + 5L(u)$, where $ED(u)$ is the difference between the number of arcs added and removed (if u were shortcut), $CN(u)$ is the number of previously contracted neighbors, $H(u)$ is the number of arcs represented by the shortcuts added, and $L(u)$ is the *level* u would be assigned to. We define $L(u)$ as $L(v) + 1$, where v is the highest-level vertex among all lower-ranked neighbors of u in G^+; if there is no such v, $L(u) = 0$.

Labeling algorithm. The preprocessing of the theoretical labeling algorithm of Abraham et al. [2] is based on *shortest path covers* (SPCs). Intuitively, an (r, k)-SPC S is a set of vertices that (1) hits every shortest path of length between r and $2r$ and (2) is *sparse*, in the sense that every ball of radius $2r$ has at most k elements from S. For a fixed parameter h (the *highway dimension* of the graph), (r, h)-SPCs exist for all r, and the greedy algorithm finds an $O(r, O(h \log n))$-SPC. The value of h is believed to be small for road networks.

The preprocessing routine computes greedy SPCs C_i for $r = 2^i,\ 0 \leq i \leq \log D$, where D is the graph diameter. For each v, it takes as a label the union over i of C_i intersected with the ball of radius $2 \cdot 2^i$ around v. As stated, the algorithm is impractical for continent-sized road networks. Greedy SPCs require many all-pairs shortest paths computations, which would take months. Furthermore, the theoretical bound on the label size ($O(k \log n \log D)$) could be in the thousands, leading to unrealistic space requirements and uncompetitive queries.

Experimental setup. Since we use actual measurements to justify our design decisions, we describe our experimental setup in advance. We implemented our algorithm in C++ and compiled it with Microsoft Visual C++ 2010. We ran our

tests on a machine with two Intel Xeon X5680 processors and 96 GB of DDR3-1333 RAM, running Windows 2008R2 Server. Each CPU has 6 cores clocked at 3.33 GHz, 6 x 64 kB L1, 6 x 256 kB L2, and 12 MB L3 cache. Preprocessing is parallelized (with OpenMP), but queries are sequential and pinned to one core.

In most experiments we report the (parallel) preprocessing time (excluding the CH preprocessing) and total space consumption in GB. For most of the paper, we evaluate queries by running 100 000 000 s–t queries (with s and t picked uniformly at random in advance) and reporting the average time. We focus on computing the *length* of shortest paths; for full path descriptions, one could apply the expansion techniques used for TNR [4], for example.

We use two input graphs taken from the 9th DIMACS Implementation Challenge [10]. The *Europe* instance, representing Western Europe, has 18 million vertices and 44 million arcs. The *USA* road network has 24 million vertices and 58 million arcs. In both cases, arc costs are 32-bit integers representing travel times. Unless otherwise mentioned, we use the Europe instance as default.

3 HL Overview

Preprocessing. Geisberger et al. [14] suggest implementing many-to-many queries by precomputing and storing the sets of vertices of the forward CH searches for a set of sources and of the reverse CH searches for a set of targets, along with the corresponding distance estimates. A query from a source to a target is done by intersecting the corresponding sets. They have not pursued this approach for point-to-point queries, probably because it looked impractical. Indeed, our sampling-based estimates for Europe show that one would need about 154 GB to store all labels (whose average size is 536). The time estimates are encouraging, however: 321 seconds to compute all labels and 3 μs for queries. To make the algorithm truly practical, however, we need several additional ingredients.

In particular, the sets visited by CH are not *strict labels*: a bound $d(w)$ stored within a label for v may actually be greater than $\text{dist}(v, w)$. As Section 4 will show, we can efficiently *prune* each label by eliminating entries with wrong distance estimates. A simple heuristic (based on stall-on-demand [14]) reduces the label size to about 133, which is already much more practical. As Section 4 will show, we can go further and remove *all* vertices whose distance estimates are not tight, making the labels strict. By combining this with ideas from the theoretical algorithm [2], we achieve labels with fewer than 85 entries on average.

Query. We now consider how to represent labels to allow efficient queries. We describe the L_f labels; the L_r labels are symmetric. A forward label $L_f(v)$ is represented as the concatenation of three elements: (1) a 32-bit integer N_v representing the number of vertices in the label; (2) a zero-based array I_v with the (32-bit) IDs of all vertices in the label, in ascending order; and (3) an array D_v with the (32-bit) distances from v to each vertex in the label. Note that vertices appear in the same order in I_v and D_v: $D_v[i] = \text{dist}(v, I_v[i])$.

Given s and t, the query algorithm must pick, among all vertices $w \in L_f(s) \cap L_r(t)$, the one minimizing $d_s(w) + d_t(w) = \text{dist}(s, w) + \text{dist}(w, t)$. Because the I_v

arrays are sorted, this can be done with a single sweep through the labels, similar to mergesort. We maintain array indices i_s and i_t (initially zero) and a tentative distance μ (initially infinite). At each step we compare $I_s[i_s]$ and $I_t[i_t]$. If these IDs are equal, we found a new w in the intersection of the labels, so we compute a new tentative distance $D_s[i_s] + D_t[i_t]$, update μ if necessary, then increment both i_s and i_t. If the IDs differ, we increment either i_s (if $I_s[i_s] < I_t[i_t]$) or i_t (if $I_s[i_s] > I_t[i_t]$). We stop when either $i_s = N_s$ or $i_t = N_t$, and return μ.

Low-level details. Implementation details are important because the fastest version of our query is less than five times slower than a random memory access.

A key aspect of the algorithm is that it accesses each array sequentially, thus minimizing the number of cache misses. Avoiding cache misses is also the motivation for having I_v and D_v as separate arrays: while we must access almost all IDs in a label, distances are only needed when IDs match. We also align each label to a cache line, which has 64 bytes in our machine.

Another practical improvement is to use the highest-ranked vertex as a sentinel by assigning ID n to it. Because this vertex must belong to all labels, it will lead to a match in every query; it therefore suffices to test for termination only after a match. In addition, we store the distance to the sentinel at the beginning of the label; this enables us to obtain a quick upper bound on the s–t distance.

We forced procedure inlining whenever appropriate (a function call takes about 150 ns, roughly the time of 3 memory accesses), and prefetch data to the L1 cache whenever appropriate. Finally, we use pointer arithmetic (instead of maintaining indices) to traverse the labels during queries.

4 Efficient HL Implementation

This section introduces techniques that make HL efficient by reducing the average label size, speeding up long-distance queries, and using compression. We also describe several lower-level improvements.

4.1 Label Pruning

We can use a fast heuristic modification (similar to *stall-on-demand* [20]) to the CH search to identify most vertices with incorrect distance bounds. Suppose we are performing a forward CH search (the reverse case is similar) from v and we are about to scan w, with distance bound $d(w)$. We examine all incoming arcs $(u, w) \in A^{\downarrow}$. If $d(w) > d(u) + \ell(u, w)$, then $d(w)$ is provably incorrect. We can safely remove w from the label, and we do not scan its outgoing arcs. This technique significantly decreases the average label size (to 133.0) and query time (to 937 ns).

We use *bootstrapping* (i.e., HL itself) to prune the labels further. We compute labels in descending level order. Suppose we have just computed the partially pruned label $L_f(v)$. We know that $d(v) = 0$ and that all other vertices w in $L_f(v)$ have higher level than v, which means $L_r(w)$ must have already been computed. We can therefore compute $\text{dist}(v, w)$ by running a v–w HL query,

using $L_f(v)$ itself and the precomputed label $L_r(w)$. We remove w from $L_f(v)$ if $d(w) > \text{dist}(v, w)$. Bootstrapping reduces the average label size to 109.6 (30.6 GB in total), and improves average queries to 812 ns. Preprocessing is slightly slower, at 580 s. The resulting labeling algorithm is strict and practical, but substantial further improvements are possible.

Note that, without bootstrapping, labels can be trivially computed in parallel, since they are independent. Bootstrapping requires greater care. We can process vertices of the same level in parallel, but must synchronize after each level, since computing the label of a level-i vertex requires access to labels at higher levels. Fortunately, road networks have only about 150 levels [7].

4.2 Label Ordering

We can assign new *internal IDs* to the vertices to change the order in which they appear in the labels; this may speed up queries or improve compression rates.

For most vertices, keeping the original *input* order seems to be a good idea. Rearranging vertices by rank or level (either ascending or descending) actually increases query times on Europe from 812 ns to more than 1100 ns. This happens because nearby vertices in the graph tend to have similar original IDs. During an s–t query, a large portion of the corresponding labels represents vertices in small regions around s and t; it is often the case that all vertex IDs in one region are larger than all IDs in the other. As a result, the query algorithm may reach the end of one label (thus stopping the search) while visiting a fraction of the other. Rearranging vertices destroys this locality and decreases query performance.

For faster queries, it is often better to keep the input order for all but the topmost (highest-ranked) k vertices, which are assigned internal IDs from 0 to $k-1$. In particular, the *top* k *input* order (in which the input order among the top k vertices is preserved), achieves query times of 769 ns with $k = 256$. The *top* k *level* order (which sorts the top k vertices by level), is slightly worse: query times are about the same as keeping the original input order (for $k = 256$). Unless otherwise stated, we use the top 256 input order.

As already mentioned, one optimization we apply to all label orderings is to assign ID $n = |V|$ to the highest-ranked vertex, which is used as a sentinel.

4.3 Shortest Path Covers

The CH preprocessing algorithm tends to contract the least important vertices (those on few shortest paths) first, and the more important vertices (those on more shortest paths) later. The heuristic used to choose the next vertex to contract works poorly near the end of preprocessing, when it must order important vertices relative to one another. This has been observed before [14]: a variant of TNR based on CH yielded worse locality filters than previous versions. We use shortest path covers to improve the ordering of important vertices. We do this near the end of CH preprocessing, when most vertices have been contracted, the graph is small, and the greedy SPC algorithm becomes feasible.

More precisely, we start by running the CH preprocessing with our original selection rule, but pause it as soon as the remaining graph G_t has only t vertices

left (we use $t = 25\,000$). We then run a greedy algorithm to find a set C of good cover vertices, i.e., vertices that hit a large fraction of all shortest paths of G_t, with $|C| < t$ (we use $|C| = 2048$). Starting with $C = \emptyset$, at each step we add to C the vertex v that hits the most uncovered (by C) shortest paths in G_t. After C is computed, we continue the CH preprocessing, but forbid the contraction of the vertices in C until they are the only ones left. This ensures the top $|C|$ vertices of the hierarchy will be exactly those in C, which are then contracted in reverse greedy order (i.e., the first vertex found by the greedy algorithm is the last one remaining).

Setting $t = 25\,000$ and $|C| = 2048$ decreases the average label size on Europe by about 20%, from 109.62 to 84.74. Query times are reduced accordingly, from 769 ns to 594 ns. Given our emphasis on query times, we use the SPC-augmented preprocessing with these parameters as default. The time to build the hierarchy increases from 3 minutes to 151 minutes, however. If this is an issue, a good compromise is to use $t = 10\,000$ and $|C| = 512$: preprocessing takes only 25 minutes, but queries are almost as fast (598 ns) and labels almost as small (85.79 entries) as with the original parameters.

4.4 Label Compression

Even after reducing the average label size from 536 to 85, we still need 23.9 GB to store all labels if we represent every vertex ID and distance as a separate 32-bit integer. For low-ID vertices, we can use an *8/24 compression* scheme: we represent each of the first 256 vertices as a single 32-bit word, with 8 bits allocated to the ID and 24 bits to the distance. (This could obviously be generalized for different numbers of bits.) For effectiveness, it pays to reorder vertices so that the important ones (which appear in most labels) have the lowest IDs. With top 256 input ordering, the space usage decreases from 23.9 GB to 20.1 GB. Because of better locality, queries also improve, from 594 ns to 572 ns.

Another compression technique we considered exploits the fact that the forward (or reverse) CH trees of two nearby vertices in a road network are different near the roots, but are often the same when sufficiently away from them, where the most important vertices appear. By reordering vertices in reverse rank order, for example, the labels of nearby vertices will often share long common prefixes, with the same sets of vertices (but usually different distances). Our compression scheme computes a dictionary of the common label prefixes and reuses them.

Given a parameter k, the *k-prefix compression* scheme decomposes each forward label $L_f(v)$ (reverse labels are similar) into a *prefix* $P_k(v)$ (with the vertices with internal ID lower than k) and a *suffix* $S_k(v)$ (with the remaining vertices).

Take the forward (pruned) CH search tree T_v from v: $S_k(v)$ induces a subtree containing v (unless $S_k(v)$ is empty), and $P_k(v)$ induces a forest F. The *base* $b(w)$ of a vertex $w \in P_k(v)$ is the parent of the root of w's tree in F; by definition, $b(w) \in S_k(v)$. (If $S_k(v)$ is empty, let $b(v) = v$.) Each prefix $P_k(v)$ is represented as a list of triples $(w, \delta(w), \pi(w))$, where $\delta(w)$ is the distance between $b(w)$ and w, and $\pi(w)$ is the position of $b(w)$ in $S_k(v)$. Two prefixes are *equal* only if they consist of the exact same triples. We build a dictionary (an array) consisting of

all *distinct* prefixes. Each triple uses 64 consecutive bits: 32 for the ID, 24 for $\delta(\cdot)$, and 8 for $\pi(\cdot)$. A forward label $L_f(v)$ has three elements: the position of its prefix $P_k(v)$ in the dictionary, the number of vertices in the suffix $S_k(v)$, and $S_k(v)$ itself (represented as before). To save space, labels are not cache-aligned.

During a query from v, suppose w is in $P_k(v)$. We have $\text{dist}(b(w), w) = \delta(w)$ and we know the position $\pi(w)$ of $b(w)$ in $S_k(v)$, where $\text{dist}(v, b(w))$ is stored explicitly. We can therefore compute $\text{dist}(v, w) = \text{dist}(v, b(w)) + \text{dist}(b(w), w)$.

On Europe, this approach reduces the space usage from 20.1 GB to 8.0 GB (with $k = 2^{16}$), for the price of a slightly longer preprocessing (502 s instead of 489 s). At 1172 ns, queries become about twice as slow.

To save even more, we use a *flexible prefix compression* scheme. Instead of using the same threshold k for all labels, it may split each label L in two arbitrarily. As before, common prefixes are represented once and shared among labels. Deciding which prefixes to keep is no longer straightforward. To minimize the total space usage, including all n suffixes and the (up to n) prefixes we actually keep, we model this as a *facility location* [18] problem. Each label is a *customer* that must be represented (served) by a suitable prefix (facility). The opening cost of a facility is the size of the corresponding prefix. The cost of serving a customer L by a prefix P is the size of the corresponding suffix ($|L| - |P|$). Each label L is served by the available prefix that minimizes the service cost. We use local search [18] to find a good heuristic solution.

The flexible approach reduces the space usage to 5.6 GB with the same query time (1170 ns), but the preprocessing time increases from 502 s to 2002 s.

4.5 Partition Oracle

We now describe an acceleration technique for long-range HL queries. If the source and the target are far apart, the HL searches tend to meet at very important (high-rank) vertices. If we rearrange the labels such that more important vertices appear before less important ones, long-range queries can stop traversing the labels when sufficiently unimportant vertices are reached.

During preprocessing, we first find a good partition of the graph into cells of bounded size, while trying to minimize the total number b of boundary vertices.

Second, we perform CH preprocessing as usual, but delay the contraction of boundary vertices until the contracted graph has at most $2b$ vertices. Let B^+ be the set of all vertices with rank at least as high as that of the lowest-ranked boundary vertex. This set includes all boundary vertices and has size $|B^+| \leq 2b$.

Third, we compute labels in normal fashion, but we also store at the beginning of a label for v the ID of the cell v belongs to.

Fourth, for every pair (C_i, C_j) of cells, we run HL queries between each vertex in $B^+ \cap C_i$ and each vertex in $B^+ \cap C_j$, and keep track of the internal ID of their meeting vertex. Let m_{ij} be the maximum such ID over all queries made for this pair of cells. We then build a $k \times k$ matrix, with entry (i, j) corresponding to m_{ij} and represented with 32 bits. Building the matrix requires up to $4b^2$ queries and concludes the preprocessing stage.

An s–t query (with $s \in C_a$ and $t \in C_b$) looks at vertices in increasing order of internal ID (as usual), but now it stops as soon as it reaches (in either label) a vertex with internal ID higher than m_{ab}—we know no query from C_a to C_b meets at a vertex higher than m_{ab}. Although this strategy needs one extra memory access to retrieve m_{ab}, long-range queries only look at a fraction of each label.

In practice, we use the PUNCH algorithm [8] to partition Europe into cells with up to $U = 20\,000$ vertices. It takes less than 3 minutes to find the partition, and 4 minutes to compute the oracle (matrix). Building the contraction hierarchy (with 2048/25K SPCs) takes about 2.5 hours. We use a top 2048 level order and 8/24 compression. Using the oracle reduces average query times from 572 ns to 357 ns. Local queries get slightly worse, mainly due to the different label ordering.

4.6 Index-Free Labels

To perform an s–t query, HL must bring two labels, $L_f(s)$ and $L_r(t)$, from memory. To locate these labels in memory, it must access the entries for s and t in an *index array*. When applying all the speed-oriented optimizations described above, these two accesses can be a significant fraction of the query time.

We can eliminate the index array as follows. We reserve c bytes in each label array (forward and reverse) for each label. We store the first c bytes of $L_f(v)$ at position $v \cdot c$ in the forward label array (the reverse case is similar); the remaining entries—if any—are stored in a third array (the *escape array*). Each label (in the label array) also stores an index to the escape array. An s–t query starts reading the label arrays directly (with no index), and continues reading from the escape array if necessary. This approach increases the memory footprint of HL (since it allocates too much space for short labels), but accelerates queries that do not access the escape array. The choice of c determines the trade-off between memory and query times.

On Europe with the oracle, queries are fastest (276 ns, from 357 ns) with $c = 512$. The total space increases very little (20.1 GB to 21.3 GB), since almost two-thirds of the labels are split. The oracle ensures we rarely have to access the escape array. Indeed, using $c = 1024$ (when only 0.2% of the labels are split) requires much more space (34.4 GB) but query times are similar (280 ns). With no oracle, query times vary from 650 ns ($c = 512$) to 479 ns ($c = 1024$).

5 Experimental Results

We consider three variants of HL. The *prefix* variant is optimized for space: it uses the flexible prefix compression scheme (with inverse rank order), an index, and the oracle. The *global* variant is optimized for random and long-range queries: it uses the oracle (with top 2048 level order), no index, and 8/24 compression. The *local* version is optimized for fast short- and mid-range queries, which are more common in practice; it uses an index but no oracle, 8/24 compression, and top 256 input order.

Table 1 compares preprocessing and random queries for all three HL variants and five previously known fast algorithms. The first is CH [14]. The second,

CHASE, is a combination of CH and arc flags [5]. The third algorithm is High-Performance Multi-Level Routing (HPML) [9]: its preprocessing uses separators to build a large number of small auxiliary graphs, and each query composes some of them appropriately to create an acyclic search graph. The fourth algorithm is transit node routing [3,4]. Long-range TNR queries consist basically of table lookups of distances between important (transit) nodes; for short-range queries, it uses CH. Finally, we consider TNR+AF [5], a combination of TNR and arc flags that reduces the average number of table lookups to less than four. Since these algorithms were tested on an older AMD machine [5,9], Table 1 shows *scaled* running times, obtained by dividing the best published times by 1.915, the factor by which our Xeon CPU is faster (based on our calibration experiments).

The table includes a (hypothetical) implementation of a Table Lookup algorithm: it precomputes all pairs of distances, reducing queries to a single lookup. Preprocessing would be fast enough on a GPU [7], but space usage is prohibitive. We use a random memory access as an estimate of its query time.

To analyze local queries, Figure 1 plots median query times against Dijkstra rank [19]. For a search from s, the Dijkstra rank of v is i if v is the i-th vertex scanned when Dijkstra's algorithm is run from s. For HL, we run 10 000 queries per rank. All times for non-HL algorithms are taken from [5,9] and scaled.

Although practical, HL preprocessing is slower than existing algorithms, considering they could be easily parallelized. TNR, in particular, is at least an order of magnitude faster in this regard (and can be improved even further [13]). This gap in preprocessing time between HL and other methods can be much smaller if slightly slower queries are acceptable, but our emphasis is on query times.

All variants of HL have faster queries than previous techniques. For random queries, HL global is about 3.5 times faster than TNR+AF and 6 times faster than TNR. Figure 1 shows that TNR is slower on short- or mid-range queries, taking 4 μs to 10 μs; HL local is an order of magnitude faster. This should also hold for TNR+AF, since arc flags only accelerate long-range TNR queries.

Table 1. Results on random queries. HL preprocessing is parallelized (others are not) with the times for building the hierarchy and computing the labels reported separately. Table Lookup preprocessing excludes copying distances from GPU to main memory.

	Europe			USA		
method	preprocessing time [h:m]	space [GB]	query [ns]	preprocessing time [h:m]	space [GB]	query [ns]
CH [5]	0:13	0.4	93 995	0:14	0.5	67 885
CHASE [5]	0:52	0.6	9 034	1:59	0.7	9 922
HPML [9]	≈12:00	3.0	9 817	≈12:00	5.1	10 078
TNR [5]	0:58	3.7	1 775	0:47	5.4	1 566
TNR+AF [5]	2:00	5.7	992	1:22	6.3	888
HL prefix	2:31 + 0:45	5.7	527	2:17 + 0:40	6.4	542
HL local	2:31 + 0:08	20.1	572	2:17 + 0:07	22.7	627
HL global	2:31 + 0:14	21.3	276	2:17 + 0:18	25.4	266
Table Lookup	> 11:03	1 208 358.7	56	> 22:44	2 293 902.1	56

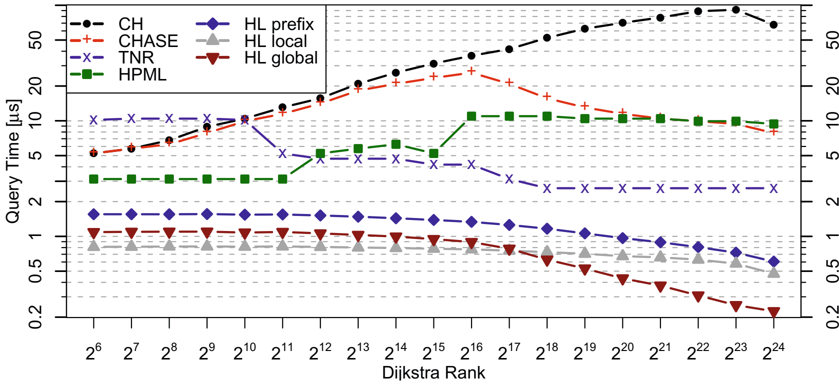


Fig. 1. Median query times on Europe for various ranges

(Unfortunately, there are no published values for short- and mid-range TNR+AF queries.) Although HL performs more operations than TNR+AF, better locality leads to fewer accesses to the main memory, which explains why it is faster. For short-range queries, the fastest previous algorithm (HPML) is four times slower than HL local and almost three times slower than HL global. (The published implementation of HPML [9] cannot handle some short-range queries, though it could easily be composed with CH.) In fact, HL global is only five times slower than Table Lookup (i.e., one random memory access) on average. For short- and mid-range queries, HL local is about 13 times slower than a random access.

Finally, we note that HL prefix needs a quarter of the space of HL global, but is only twice as slow, which is fast enough to outperform previous methods.

6 Concluding Remarks

We presented Hub Labels (HL), a labeling algorithm to compute exact point-to-point shortest paths in road networks. HL combines elements from a theoretical algorithm with contraction hierarchies. With careful engineering, HL is significantly faster than the best previous approaches for queries of all ranges. Some of our techniques may help accelerate other methods as well; in particular, a variant of our partition oracle could be used as a locality filter for TNR.

Our results show that road networks admit smaller labelings than the bounds of [2] suggest. It would be interesting to prove better bounds. Finding better SPCs or CH orderings, or faster algorithms to compute them, could improve HL even further by reducing the average label size. In particular, one would like a fast algorithm to approximate the smallest labeling (the method in [6] is impractical for large networks). Reducing the space usage of HL is also desirable, as are extensions to time-dependent and other augmented networks.

References

1. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. Technical Report MSR-TR-2010-165, Microsoft Research (2010)
2. Abraham, I., Fiat, A., Goldberg, A.V., Werneck, R.F.: Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In: SODA, pp. 782–793 (2010)
3. Bast, H., Funke, S., Matijevic, D.: Ultrafast shortest-path queries via transit nodes. In: Demetrescu, C., et al. (eds.) [10], pp. 175–192
4. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Shortest-Path Queries in Road Networks. In: ALENEX, pp. 46–59. SIAM, Philadelphia (2007)
5. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. ACM Journal of Experimental Algorithmics 15(2.3), 1–31 (2010)
6. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. 32 (2003)
7. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.: PHAST: Hardware-Accelerated Shortest Path Trees. In: IPDPS. IEEE, Los Alamitos (2011) (to appear)
8. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph Partitioning with Natural Cuts. In: IPDPS. IEEE, Los Alamitos (2011) (to appear)
9. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-Performance Multi-Level Routing. In: Demetrescu, C., et al. (eds.) [10], pp. 73–92
10. Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.): The Shortest Path Problem: Ninth DIMACS Implementation Challenge. DIMACS, vol. 74. AMS, Providence (2009)
11. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271 (1959)
12. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance labeling in graphs. J. Algorithms 53(1), 85–112 (2004)
13. Geisberger, R.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. Master's thesis, Karlsruhe University (2008)
14. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
15. Goldberg, A.V.: A Practical Shortest Path Algorithm with Linear Expected Time. SIAM Journal on Computing 37, 1637–1655 (2008)
16. Hilger, M., Köhler, E., Möhring, R.H., Schilling, H.: Fast Point-to-Point Shortest Path Computations with Arc-Flags. In: Demetrescu, C., et al. (eds.) [10], pp. 41–72
17. Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D.: Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In: ALENEX, pp. 36–45 (2007)
18. Resende, M., Werneck, R.F.: A Fast Swap-based Local Search Procedure for Location Problems. Annals of Operations Research 150, 205–230 (2007)
19. 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)
20. Schultes, D., Sanders, P.: Dynamic Highway-Node Routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
21. Thorup, M., Zwick, U.: Approximate Distance Oracles. Journal of the ACM 52(1), 1–24 (2005)

Hierarchy Decomposition for Faster User Equilibria on Road Networks*

Dennis Luxen and Peter Sanders

Karlsruhe Institute of Technology (KIT), Germany
{luxen,sanders}@kit.edu

Abstract. The road traffic of an entire day for a certain region can be understood as a flow with sources and sinks on the road network. Traffic has the tendency to evade regularly clogged roads and other bottlenecks, especially with modern on-board navigation devices that are able to interpret traffic information. Assuming perfect knowledge for all drivers, one might suspect traffic to shape itself in a way such that all used routes between any two points on the road network have equal latency. Although these traffic patterns do not or very seldom occur in real life, they are a handy tool to predict the general traffic situation. For small networks, these patterns can be easily computed, but road networks that model entire countries are still a hurdle, because Dijkstra's algorithm does not scale. Thus the known techniques have only been applied to either small networks or small extracts of a much larger network. We solve this problem for country sized road networks by combining a gradient descent method to the problem with current research on fast route planning by exploiting the special properties of a routing algorithm called Contraction Hierarchies. The computation of the gradient needs a large number of shortest paths computations on the same weighted graph, which means that the expense for preprocessing can be amortized if the number of shortest paths computations is sufficiently large. This leads to dramatic overall speedup compared to running Dijkstra for each demand pair. Also, our study shows the robustness of Contraction Hierarchies on road networks at equilibrium state.

1 Introduction

Traffic is often seen as a mere stream of cars. Consider the following picture. On a typical day of work traffic flows from the suburbs into inner cities in the morning and back from it in the evening. Or on national holidays a stream of cars and buses flows perhaps to resort towns or recreation areas close to the metropolitan areas. Naturally, some roads are more crowded than others since traffic is not equally distributed over the road network. As a matter of fact, traffic has a natural tendency to shift itself to alternatives if it is more convenient for a driver to take another route. Drivers seek to minimize travel time (or any other metric) and can be understood to act as selfish agents. They switch to better routes if they become aware of it. Assuming all drivers have full knowledge one is interested in how the traffic distributes itself over the road network. This problem is known as the traffic assignment problem and is a major application

* Partially supported by DFG grant SA 933/5-1.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 242–253, 2011.
© Springer-Verlag Berlin Heidelberg 2011

in the field of transportation planning. Visually speaking, it is the process that finds edge latencies in a road network that are the result of many individuals competing for transportation. We assume travelers to take least cost or (under some metric) shortest paths between their origins and destinations. The problem at hand has been the subject of research since the early 1950s. Wardrop's [31] first principle states the properties for a so-called *user equilibrium state*, which resembles the natural tendency of traffic to take a way of least resistance.

Definition 1 (Wardrop's User Equilibrium). *A set of flows along the edges of a road network is said to be in a user equilibrium state (UE) when the two conditions of the following definition are met.*

1. *If two or more paths between the origin s and the destination t are actually traveled, then the cost of each path between s and t actually used must be the same.*
2. *There does not exist any path between s and t that is of less cost and unused.*

Finding a traffic pattern for which the above conditions hold is called *traffic assignment problem* (TAP). Solutions to this problem have a wide range of applications, for example in transportation management or in traveler information systems. Also, the real-time computation of equilibria states can be used as traffic forecasts and for traffic steering. Basic traffic jam avoidance is a feature of nowadays navigation devices. Unfortunately, this feature is not as developed as it is advertised.

Consider the following example. A traffic jam is reported for a certain highway and drivers on that highway are advised to leave their route by switching to an alternate road nearby. Since many drivers leave the highway, the road nearby is also clogged. This is not just an academic example, but happens every day. Germany's biggest automobile club ADAC reports in a large scale study [29] that most towns close to a highway suffer from increased pass-through traffic because of jam evaders. Routing on a road network that is at equilibrium is said to be a good estimate of routes that make not only economical sense but also are perceived as good alternatives to a clogged route. Today's jam evading features of navigation devices is limited. ADAC also reports a field study [6] that shows the inferiority of current approaches for traffic jam evasion. Not only the current traffic situation has to be considered to give better guidance around traffic jams, but also how the traffic will evolve.

Travelers on a road network are said to be non-cooperating. The state of the equilibrium is the aggregate result of individual decisions and therefore the name *user equilibrium*. It is generally assumed that under equilibrium conditions all used routes for the same origin destination pair have same costs, e.g. equal travel time. Also, unused routes between any origin destination pair have higher costs than used ones. Travelers are free to switch routes if there is a better one than the current. The traffic distributes itself in a way that no traveler can lower its path cost unilaterally by switching to a cheaper path. This is the case at equilibrium, because by definition there is simply no such path. Note that this equilibrium state can also be modeled as a Nash Equilibrium [25].

A general behavioral assumption in the field of transportation science is that each traveler or vehicle in a road network will take a path that has least cost (or is at least perceived as such). It is further assumed that travel time is the most significant utility for

route choice. We recognize the over-simplification of this model, but direct the reader to the literature on empirical research of route choice, i.e [23].

The remainder of this paper is organized as follows. First, we look at the relevant literature in Section 2. We introduce the basic algorithms and data structures that we use and explain how they solve the problem at hand in Section 3. Section 4 explains how our approach can be tightly integrated into Contraction Hierarchies, which is a well-known speedup technique to Dijkstra's algorithm. Second, we present an experimental evaluation in Section 5 that shows the performance of our approach. The method is applied to a graph that models the entire road network of Belgium and Germany. Section 6 summarizes the results and presents future directions of research.

2 Related Work

The TAP has been studied for more than 50 years. The first mathematical formulation is generally attributed to Beckman et al. [4]. It was first given in 1956 and models the TAP as an optimization problem.

The method of choice to solve this problem is the Frank-Wolfe algorithm [18], which is also known as the *convex combinations algorithm*. It was originally invented to solve quadratic programming problems. Over the years it has been applied to the traffic assignment problem, mainly because of its rather simple structure. Occurrences in the literature go back to the late 1960s [7,19]. The major advantage of the Frank-Wolfe algorithm (besides its simplicity) is its low memory consumption. For example, it does not save any information on computed routes. It only counts the volume of traffic on each individual street segment. This was considered a major advantage in the early days of computation, because of limited memory capabilities. The algorithm alternates between an assignment phase of the traffic demand and a minimization step to numerically approximate edge flows.

The textbook of Sheffi [26] gives an overview of the first three decades of research between 1950 and 1980. Most of the solution techniques described are still in use by practitioners today. Usually they are applied to road networks of small and medium size up to several hundred or a few thousand edges and often only on sparse subsets of highway networks which are much smaller than the full road network.

There are several publications that focus on speeding up convergence of solving the traffic assignment problem by modifying the way traffic flow is distributed during the computation. Gentile [14] proposes an algorithm that seeks a deterministic equilibrium for the local route choice of users directed toward a same destination at every node. Bar-Gera [1] presents an algorithm to compute the UE by paired alternative segments. If flow between two nodes splits into separate sub-paths then flow is shifted proportionally.

A completely different model to solve the traffic assignment problem is to apply game theory. Rosenthal [24] was the first to consider the problem by a game theoretic approach. A so-called congestion game is defined by a set of players that compete for one or more shared resources. It is said to be symmetric if all players chose among the same set of strategies. Fabrikant et al. [12] show that any symmetric congestion game can be solved in polynomial time. Relating to our case the players are travelers that compete for roads and seek to minimize travel expenses. Edge latencies, i.e. the time time it takes to traverse a road segment, are defined to be nonnegative, continuous and

nondecreasing functions of the amount of travelers on that edge. Consider building up a given traffic flow on a road from zero flow, one infinitesimal flow path at a time. The potential function is obtained by integrating the latency experienced by each infinitesimal flow path, using edge latencies that were in effect at the moment it was routed. This potential can be easily optimized to a (local) minimum by allowing players to switch their strategy, which is a shortest route in this case. These switches are called selfish steps. Consider a move of one of the players to a better route. Any local optimum corresponds to the conditions stated in Definition 1. It is easy to see that the potential is lowered and that it can be brought to a minimum by subsequent switches until no switch to an improved route for any player is possible.

Kirschner et al. [17] apply book keeping heuristics to avoid many path computations and subsequently speed up the rate of convergence on networks with less than a few thousand nodes and edges. For an excellent survey over the literature for congestion games and algorithmic game theory in general see the textbook of Nisan et al. [20] and especially Roughgarden's seminal work on the Price of Anarchy in routing games [25]. Note that the game theoretic approach prohibits any precomputation fixed set of edge costs, because edge weights change with every move. A single selfish step might change the edge weights enough to invalidate the preprocessed data structures and preprocessing the network for a single query is out of the question.

The application of the Frank-Wolfe algorithm and also the game theoretic solution need a method of path finding. Plain solutions spend virtually all of the computational effort in path finding. Unfortunately, Dijkstra's algorithm does not scale well on large road networks with millions of nodes and edges. For large scale applications [21] this is simply unacceptable. Therefore speedup techniques for point-to-point queries with Dijkstra's classic algorithm have been the focus of numerous publications before. For surveys on the literature and combinations of several methods see [11,3].

Contraction Hierarchies (CH) [13] is a very successful speedup technique that has the advantage of combining a simple algorithmic concept and very good speedups. CH performs precomputation on a directed graph $G = (V, E)$, with edge weight function $c : E \to R_+$. All nodes are ordered by importance and the CH is constructed by contracting the nodes in the above order. Contracting a node u means removing u from the graph without changing shortest path distances between the remaining (more important) nodes. Contracted nodes are bypassed and replaced by so-called shortcut edges. Given a pair of of adjacent edges (v, u) and (u, w), the shortest path P between v and w avoiding u is computed. Only if the length of P is longer than the length of the path $\langle v, u, w\rangle$, a shortcut edge between v and w is necessary with the combined weight $c(v, u) + c(u, w)$. The resulting graph can be queried by a bidirectional Dijkstra to find shortest paths. We refer the interested reader to the publication of Geisberger et al. [13] for an in-depth explanation of the node ordering and proofs of correctness. There have also been reports on combinations of several distinct speedup techniques [3].

To the best of our knowledge there is no publication that reports on directly exploiting special properties of any speedup technique to augment traffic assignment computations. Also, we are not aware of equilibria computations for large networks with significantly more than a few hundred or thousand street segments [15,1,14].

3 Problem Formulation

We model a road network as a graph $G = (V, E)$. V is a set of nodes and $E \in V \times V$ is a set of directed edges or less formally the set of street segments. Each edge carries a certain amount of traffic that we call flow. Each edge e is labeled with an edge weight that is the result of the flow f_e on e and edge cost function C_e. Given $|V|$ nodes in a network, let nodes $1, \ldots, p \leq |V|$ be a subset of nodes which are either origin or destination of a so-called demand set.

Generally, we view the nodes of the graph as the places where traffic passes-by, enters or leaves the system. We define the set of demands D as a set of triples (i, j, k), where $i, j \in V$ and $k \in \mathbb{N}$. The nodes i and j indicate origin and destination nodes and k the number of units that demand to flow between these nodes. Flow on a certain road segment is said to be the ratio of the current and maximum number of vehicles on that segment at a given average speed.

Optimization Problem. In [26] it has been shown that the traffic assignment problem can be solved as a minimization problem. The objective function of the underlying optimization problem is based on total edge flows and the resulting edge weights. Consider ω to be the flow on an edge. The function is defined by the sum over the change of all edge weights

$$\min(\mathbf{z}) = \sum_{e \in E} \int_0^{f_e} C_e(\omega)\, d\omega$$

with the constraint, that the sum over all observed flows between any two nodes equals the total demand between those nodes. This minimization problem can be solved by applying the Frank-Wolfe algorithm [18,26]. In each step of the algorithm the approximation of the solution is replaced by a new approximation that is obtained by gradient descent towards the optimum.

Initialization and Iterative Improvement. The initialization is an all-or-nothing assignment of the demand set where each demand is assigned to the edges of the shortest paths using free flow speed on the edges. In other words, travelers choose the routes that would be best if they were the only travelers on the road network. These free flow usages are counted and edge weights reevaluated w.r.t. the flow on the edges and these edge weights are taken as the initial solution X^0. In each iteration a subsequent assignment Y^i is computed and combined with the previous solution to get a better approximation.

More formally, the n-th iteration starts with an update of edge weights by evaluating $C_e(f_e)$ for each edge. Note that the edge flow vector $F^n = (f_1^n, \ldots f_{|E|}^n)$ is the result of the previous iteration. Next, an all-or-nothing assignment distributes the so-called auxiliary flow $Y^n = (y_1^n, \ldots y_{|E|}^n)$ on the network. The new approximation

$$X^{n+1} = X^n + \alpha^n \cdot (Y^n - X^n)$$

is obtained by computing a scaling factor α^n that is feasible in the current iteration. Note that computing Y^n is straight-forward and X^n is known from previous iteration. We solve

$$\alpha^n = \min_{0 \leq \alpha \leq 1} \sum_e \int_0^{f_e^n + \alpha(y_e^n - f_e^n)} C_e(\omega) d\omega$$

at each iteration. Since we know the derivative of the function, we can solve that step with a search strategy to find the minimum. This is also known as line search.

The search for α^n is solved approximately with a certain error threshold by applying the bisection method of Bolzano, which approximates the zero of a continuous function by binary search. For any given interval $[a, b]$ and $c = (b + a)/2$ we examine if our solution is either in $[a, c]$ or $[c, b]$ and decent recursively until we have reached a certain accuracy. The method is particularly easy to implement.

The series of solutions $X^i, i > 0$, is known to converge to the solution of the traffic assignment problem. Again, we refer the reader to the textbook of Sheffi [26] for in-depth explanations and for the correctness of the method.

Edge Cost Functions. If the travel time between any two nodes was a constant independent of the flow in between then we could solve the problem easily. It would suffice to compute the shortest path for each element of the demand set. Of course, this view neglects reality and the effect that flow, or in other words dense traffic, has to the average speed on a road segment. The denser the traffic gets the more careful drivers have to be not to cause an accident by running into a decelerating car in front. Likewise the denser the traffic the more cars are affected by one's own driving maneuvers [28].

To model the situation more realistically the edge cost function has to be increasing, contiguous and non-linear. Several good edge cost functions have been proposed. A simplified function is the Bureau of Public Roads [8] function (BPR). This function was derived from empiric observation and takes length, speed limit and capacity as parameters. Although it is easy to compute its curves are not asymptotic to any maximum capacity value, which is in stark contrast to reality. To overcome this shortage Davidson [9] proposed a function family that is based on queuing theory. It is defined as

$$t_e = t_e^0 \cdot \left[1 + J \cdot \frac{x_e}{c_e - x_e} \right]$$

where t_e^0 denotes the travel time at 0 usage, c_e the capacity of the street segment and x_e the current usage. J is a tuning parameter to control the shape of the curve.

Other classes of road functions have been proposed, e.g. the class of *conical volume-delay functions* [27]. For an earlier survey on edge cost functions see [5]. But on the other hand the Davidson function models basic relationships between usage and resulting travel times and it is easy to compute. We set J to 0.25 throughout this paper, which is a common value among practitioners.

Convergence Criterion. Convergence can be based on a number of criteria. Clearly, one would like to stop once the edge weights do not change any more between iterations. The easiest choice is to stop after a fixed number of iterations, but this entirely neglects solution quality. A natural choice would be to use the change of the objective function as convergence test. This might be misleading, since the lengths of individual paths might differ significantly while the sum of of the lengths is relatively stable. Therefore, the stopping criterion is based on how much the path length for each demand differs between two iterations.

$$\max_{d \in D} abs \left(\frac{\mu^n(d) - \mu^{n-1}(d)}{\mu^{n-1}(d)} \right)$$

where D is the set of demands and $\mu^n(d)$ is the length (cost) of the path in iteration n for demand d. This stopping criterion indicates the quality of the approximation of the equilibrium much better from a behavioral point of view than a simple sum of all edge weights. Furthermore it ensures that the computation is only stopped once the weights of all edges have settled down.

A naive implementation of the optimization algorithm is technically easy and straight-forward with any algorithm that computes shortest path, i.e. Dijkstra's Algorithm. A more efficient approach will be explained in the next Section.

4 Integration into Contraction Hierarchies

The Frank-Wolfe algorithm needs a number of shortest path computations on the same weighted graph. Consider a shortest path that is computed by the bidirectional CH query on the search data structure consists of shortcuts. Although the length of any shortest path is optimal, it has to be unpacked to get the edges of the original graph. Usually, unpacking is done by a recursive method. The edges of the packed path are pushed onto a stack and while the stack is non-empty an edge is popped. If it is a shortcut then the two edges building that shortcut are pushed onto the stack. Otherwise the popped edge is inserted into the resulting unpacked path. The recursive unpacking runs fast in time linear to the length of the unpacked path. When compared to a plain Dijkstra algorithm using a CH black box with path unpacking is still several orders of magnitude faster. We can do better with the following method, which has a running time independent of the size of the demand set. The method is not recursive and relies only on the number of shortcuts in the hierarchy.

If paths are unpacked at each time they get computed then the heavily traversed edges would be touched many, many times to increase usage counters. We modify the CH path computation to do a *hierarchy decomposition* in which each shortcut is unpacked only once in a certain order. At first, we do not unpack the paths at all, but count the flows on edges without unpacking shortcuts. To do so, each original and shortcut edge is equipped with a counter to record the number of times it is part of a shortest path. This number is counted during the path computation. It can be done easily, since each path consists of a few shortcuts only. Since we do not need to keep track of the routes actually chosen by travelers, we just count the number of times each individual edge is used. After all paths have been computed the hierarchy is decomposed by unpacking all shortcuts and assigning the load of the shortcut to edges that lie underneath. See Figure 1 for an illustration of the process of hierarchy decomposition. The only prerequisite to the correctness of this approach is to decompose the shortcuts in the

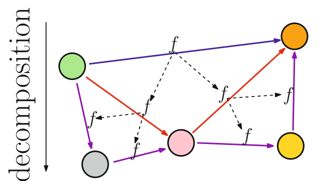


Fig. 1. Flow f is distributed through dashed edges to all underlying edges of a shortcut

opposite order in which they were created during construction of the hierarchy. It is obvious from the order of decomposition (and from the fact that the CH data structure is a directed acyclic graph) that no shortcut needs to be unpacked more than once and no edge usage is lost. The order of shortcut creation is easy to record during preprocessing and takes up a negligible amount of space only. The reverse order of insertion defines the order of the decomposition. Note that the order can also be determined by a topological search.

There exist speedup techniques to Dijkstra's algorithm that support dynamic updates. But they are not feasible options here, because the number of edge weight changes is too high. Our resulting algorithm performs very well as we will see in Section 5. The additional space overhead for shortcut order and edge usage counters is more than bearable on a current desktop computer.

5 Experimental Evaluation

We implemented our algorithm and data structures in C++ using GCC v4.3.2 compiler and full optimizations. All tests were done on a single core of an AMD Opteron 8350 CPU running at 2.0 GHz. The machine is equipped with 64 GB of RAM running Linux kernel version 2.6.27. The evaluation was done on road networks of Belgium and Germany. The Belgian network consists of 463 514 nodes and 1 093 454 edges whereas the German network consists of 4 378 446 nodes and 9 574 254 million edges. Both have been made available by PTV AG[1] for scientific use. We stick to travel time as edge cost metric throughout this paper. For each road segment length and the respective out of 13 road categories are available. Free flow speeds have been derived from category and length of an edge. Capacity is implied by the category of the edge, which is an oversimplification, but unavoidable because of lack of data. Travel times were computed according to the Project OSRM car speed profile [22].

We pregenerate randomized lists of origin destination pairs, also called *the set of demands* or *demands* for short, for the test cases of Section 5. To the best of our knowledge we are not aware of any high resolution trip generation algorithms coming from transportation science that covers entire countries. A simple trip generation model is presented that generates traffic demand that is realistic enough to show the validity of the technical approach.

During a personal conversation with an ADAC representative we were told that the distances actually traveled are geometrically distributed with an expected distance of 40 kilometers. We conjecture that the population density correlates strongly with the density of a road network. We choose the starting points uniformly and at random from the set of all nodes. Since we know the distribution, we draw a geometrically distributed distance. A ball is grown around each starting node s using a unidirectional Dijkstra Search and when an edge is relaxed we check the distance its end node has from the source. If the distance of the end node is equal or more than the travel distance that was drawn before, we accept the node as the target t of s and insert the triple $(s, t, 1)$ into the demand set. Each demand is given equal weight.

[1] http://www.ptv.de

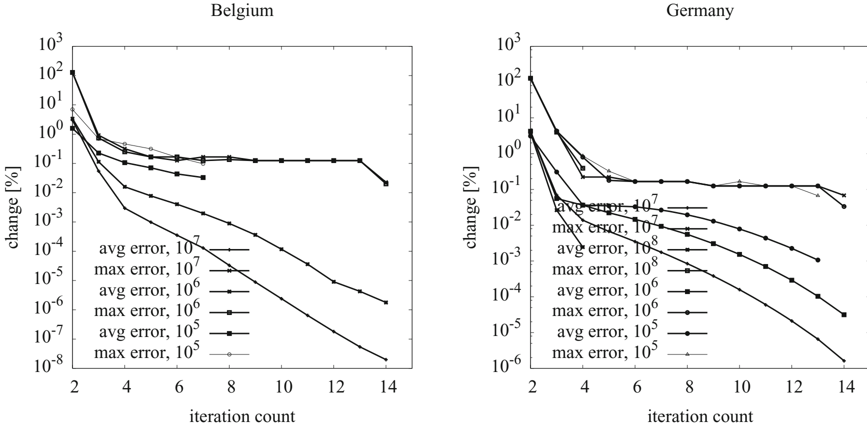


Fig. 2. Experimental results for the Belgian (left) and German (right) road network

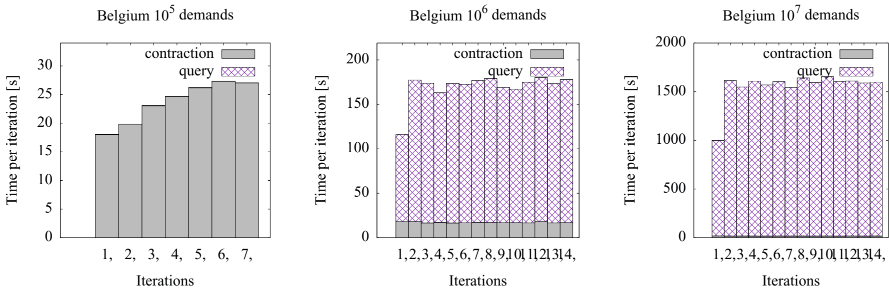


Fig. 3. Running times for Belgian network

We pregenerated lists of $10^5, 10^6$ and 10^7 demands for the network of Belgium and Germany and in addition also a list of 10^8 demands for Germany to reflect the larger size of the road network. We computed the user equilibria for all demand sets on the respective graphs. The line search approximation parameter was set to 10^{-10} and the dampening factor J of the Davidson edge cost function was set to 0.25.

See Figure 2, 3 and 4 for the numerical results of the experiments. The stopping value is quickly approached in each of the experiments. We observe that the stopping value of a maximum error of less than 0.001% is approached for each of the demand sets in a similar way while the average error drops significantly with larger demand set sizes. The larger the graph and the demand set the more evident this phenomenon gets. The larger the numbers of queries the less important preprocessing and the road network size gets, which can be seen in Figures 3 and 4. We observe that the preprocessing dominates the query phase so strongly on the Belgian test set with 10^5 demands that they are not visible in the plot at all. Likewise, the preprocessing time for the 10^8 case on the German data set is not visible. We conclude that preprocessing times are more than bearable for sufficiently large demand sets.

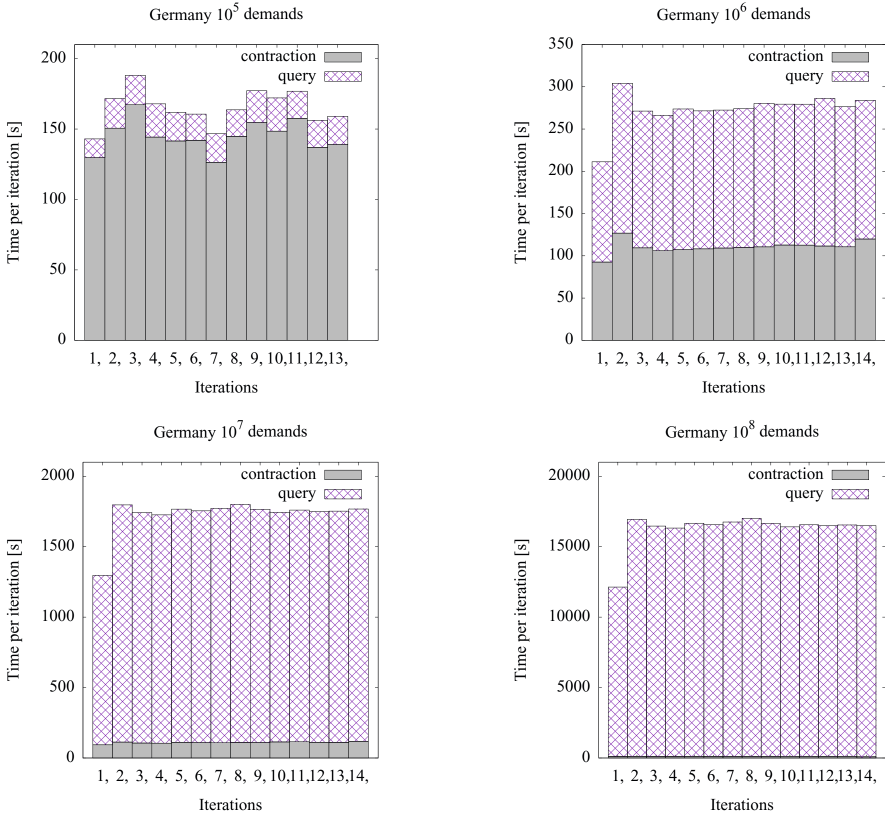


Fig. 4. Running times for German network

The traffic assignment changes the edge weights of the underlying graph. This is a direct consequence of Wardrop's User Equilibrium from Definition 1. Hence under equilibrium state all used routes for a certain origin-destination pair have equal travel times. This flattens the natural hierarchy of the road network that is exploited during the contraction phase. Thus the preprocessing takes longer, because it is harder to decide if a certain shortcut is needed or not. Less shortcut edges can be omitted from the search data structure, because for many shortcuts there is now a shortest path that actually lies on it. Likewise, the query times rise. The effect is most obvious in the first two iterations when the most changes occur in the edge weight. We observe CH to be robust on road networks at equilibrium state.

Note we also implemented a simpler iterated all-or-nothing assignment. The method starts with a feasible flow on the network. Then edge costs are recalculated for the flow, which is observed on each edge. The flow is reassigned to the changed network and the process is reiterated until a specified number of iterations is completed. We did not observe any convergence with this technique even for large numbers of iterations. In contrast, we observed oscillation of route choice and quickly deemed the approach infeasible. Likewise, an incremental loading where a subset of the demands is assigned proved infeasible as well. Again, convergence did not occur.

6 Conclusions

We presented an application of CH as a building block for a large scale optimization problem. Our algorithm exploits the special properties of the search data structure of CH which enables us to solve the problem with better efficiency than pure path computation with unpacking of each computed path. We showed the feasibility of large scale traffic assignment on graphs that cover entire countries and observe the robustness of CH. Query times could benefit from incorporating incorporating further algorithmic techniques. For example, a combination of CH and Arc-Flags [3] can now be preprocessed within a few minutes [10] and delivers single-digit query times. Also, Christian Vetter has already implemented a parallel CH variant [30] that could be adapted for hierarchy decomposition. We are working to extend our research to time-dependent road networks, multiple cost functions and also distributed computation. CH have already been adapted to time-dependent road networks [2] and there has been a distributed implementation recently [16] that could be used to speed up the preprocessing phase even further. Finding the right modeling of the time-dependent traffic assignment problem is an interesting question on its own.

References

1. Bar-Gera, H.: Traffic assignment by paired alternative segments. Transportation Research Part B (2010)
2. Batz, V., Delling, D., Sanders, P., Vetter, C.: Time-Dependent Contraction Hierarchies. In: Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX 2009), pp. 97–105. SIAM, Philadelphia (2009)
3. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In: ACM JEA special issue for WEA (2008)
4. Beckmann, M., McGuire, C.B., Winsten, C.B.: Studies in the economics of transportation. Yale University Press, New Haven (1959)
5. Branston, D.: Link capacity functions: A review. Transportation Research 10(4), 223–236 (1976)
6. Brieter, K., Eicher, C.C., Haart, V., Vigl, M.: Mit dem navi sicher in den stau. ADAC Motorwelt 3, 56–59 (2010)
7. Bruynooghe, M., Gilbert, A., Sakarovitch, M.: Une methode d'affectation du trafic. In: Proc. 4th Internat. Sympos. Theory Road Traffic (1969)
8. Bureau of Public Roads: Traffic Assignment Manual. U.S. Dept. Of Commerce, Washingtion D.C (1964)
9. Davidson, K.B.: A flow travel time relationship for use in transportation planning. Proceedings of the Australian Road Research Board 3, 183–194 (1966)
10. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.: Phast: Hardware-accelerated shortest path trees. In: 5th International Parallel and Distributed Processing Symposium, IPDPS 2011 (2011)
11. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In: Lerner, J., Wagner, D., Zweig, K.A. (eds.) Algorithmics of Large and Complex Networks. LNCS, vol. 5515, pp. 117–139. Springer, Heidelberg (2009)
12. Fabrikant, A., Papadimitriou, C., Talwar, K.: The complexity of pure nash equilibria. In: Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing, STOC 2004, pp. 604–612. ACM, New York (2004)

13. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
14. Gentile, G.: Linear user cost equilibrium: a new algorithm for traffic assignment. submitted for publication in Transportation Research B (2010)
15. Jayakrishnan, R., Tsai, W.T., Prashker, J.N., Rajadhyaksha, S.: A faster path-based algorithm for traffic assignment. Transportation Research Record (1994)
16. Kieritz, T., Luxen, D., Sanders, P., Vetter, C.: Distributed time-dependent contraction hierarchies. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 83–93. Springer, Heidelberg (2010)
17. Kirschner, M., Schengbier, P., Tscheuschner, T.: Speed-up techniques for the selfish step algorithm in network congestion games. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 173–184. Springer, Heidelberg (2009)
18. Marguerite, F., Wolfe, P.: An algorithm for quadratic programming. Naval Research Logistics Quaterly 3, 95–110 (1956)
19. Murchland, J.: Road network traffic distribution in equilibrium. Mathematical Models in the Social Sciences (1979)
20. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V. (eds.): Algorithmic Game Theory. Cambridge University Press, Cambridge (2007)
21. Patriksson, M.: The Traffic Assignment Problem: Models and Methods. VSP, Utrecht, The Netherlands (1994)
22. Project OSRM, http://project-osrm.org
23. Ramming, S.M.: Network Knowledge and Route Choice. Ph.D. thesis, Massachusetts Institute of Technology (February 2002)
24. Rosenthal, R.W.: The network equilibrium problem in integers. Networks 3, 53–59 (1973)
25. Roughgarden, T.: Selfish Routing and the Price of Anarchy. MIT Press, Cambridge (2005)
26. Sheffi, Y.: Urban Transportation Networks: Equlibrium Analysis with Mathematical Programming. Prentice-Hall, Inc., Englewood Cliffs (1982)
27. Spiess, H.: Technical Note–Conical Volume-Delay Functions. Transportation Science 24(2), 153–158 (1990)
28. Sugiyama, Y., Fukui, M., Kikuchi, M., Hasebe, K., Nakayama, A., Nishinari, K., Ichi Tadaki, S., Yukawa, S.: Traffic jams without bottlenecks – experimental evidence for the physical mechanism of the formation of a jam. New Journal of Physics 10(3), 033001 (2008)
29. Unknown Authors: TMC-Stauumfahrung:Verkehrsprobleme durch Stauverlagerungen? Tech. rep., ADAC e.V (2010)
30. Vetter, C.: Fast and Exact Mobile Navigation with OpenStreetMap Data. Master's thesis, Karlsruhe Institute of Technology (2010)
31. Wardrop, J.G.: Some theoretical aspects of road traffic research. Proceedings of the Institute of Civil Engineers 1, 325–378 (1952)

Matching in Bipartite Graph Streams in a Small Number of Passes*

Lasse Kliemann

Institut für Informatik
Christian-Albrechts-Universität zu Kiel
Christian-Albrechts-Platz 4
24098 Kiel
lki@informatik.uni-kiel.de

Abstract. We consider the maximum-cardinality matching problem in bipartite graphs. The input graph $G = (V, E)$ is not available for random access, but only as a stream, and random-access memory is limited to storing $\Theta(n)$ edges at a time, $n = |V|$. The number of passes over the input stream required to achieve the desired approximation is an important measure. It was shown by Eggert et al. (2009, 2011) that a $1+1/k$ approximation can be computed in $O(k^5)$ passes, independently of the input size. In this work, we present a new algorithm with the same approximation guarantee of $1 + 1/k$, but show experimentally that it requires two orders of magnitude fewer passes. The proven bound on the number of passes is $O(kn)$. This bound depends on the input size, and so in principle is inferior to $O(k^5)$. But we emphasize that in experiments, we do not find any correlation between theoretical bounds and actual performance: for all algorithms the number of passes observed in experiments is far below the corresponding theoretical bound. The most interesting insight comes from an experimental comparison of the previous and the new algorithm: e.g., for $k = 9$, the new one never needed more than 94 passes, even for instances with up to 2×10^6 vertices, whereas the previous one went up to more than 32 000 passes. Our main new technique is aimed at making the most out of each pass: we maintain a complex structure, using trees, for building augmenting paths.

Keywords: bipartite graph matching, massive graphs, semi-streaming algorithms, approximation schemes.

1 Introduction

Streaming and Matching – Previous Work. Let $G = (V = A \dot{\cup} B, E)$ be a bipartite, undirected graph with $n := |V|$ vertices and $|E|$ edges. We aim to find a matching of large cardinality in G, i.e., a large subset of the edges $M \subseteq E$ such

* Permanent ID of this document: dda51148-ac5b-4655-9c4f-e01f26511235
This version is dated 2011-02-28.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 254–266, 2011.
© Springer-Verlag Berlin Heidelberg 2011

that $e \cap e' = \emptyset$ for all $e, e' \in M$ with $e \neq e'$. A matching with maximum cardinality is called an *optimal matching* or a *maximum matching.* It is well known that this problem can be solved to optimality in polynomial time.

Assume now that the graph is stored in such a way that we cannot do the usual queries efficiently, like asking whether two vertices v and w are adjacent or requesting the neighborhood of a particular vertex. The only access to the graph is by doing a *pass.* A pass means that each edge is presented to us exactly once and in arbitrary order. In other words, the graph is stored on a device that only allows sequential access. There is also a fast random-access memory available, but it is of size $\Theta(n \log n)$ bits, i.e., we can store $\Theta(n)$ edges at a time, not more. When $|E| = \Omega(n^2)$, then this is not enough to store the whole graph. This is called the *semi-streaming model* [8]. The "streaming" term refers to the way the graph is presented. The "semi" attribute refers to the amount of random access memory available; in the "streaming model", memory is limited to $\Theta(\log n)$, which is too restrictive for many graph problems [4].

Clearly, we can realize a sort of "random access" to the graph on the basis of a streaming setting: to determine whether v and w are adjacent, do a pass and note whether $\{v, w\}$ occurs or not. We can also collect the neighborhood of a certain vertex in one pass. However, operating in such a manner can lead to a huge number of passes and hence be inefficient. The common requirement is that the number of passes is independent of the problem size, i.e., independent of n and $|E|$ (but it may depend on approximation parameters). Unfortunately, many algorithms are designed around adjacency and neighborhood queries, including very simple ones like BFS and DFS, which are used in many matching algorithms. Also minimum-degree heuristics (see, e.g., [6]) for matching require neighborhood information: after we have added $\{v, w\}$ to the matching, the degrees of all neighbors of v and w have to be decremented.

Many matching algorithms work by finding and eliminating augmenting paths. Repeatedly finding and eliminating augmenting paths will, in $O(n)$ steps, lead to an optimal matching. An augmenting path can be found by performing a variant of BFS starting at the free vertices (i.e., vertices not contained in any edge of the current matching) of one of the partitions of the bipartite graph. A more sophisticated approach finds a set of pairwise vertex-disjoint (shortly: disjoint) paths and then eliminates them all together. This is the basic idea for the algorithm by Hopcroft and Karp [5] running in $O(n^{2.5})$ time. Unfortunately, with neither approach we can guarantee an appropriate bound on the number of passes required, when in a streaming situation. McGregor [7] suggested resorting to approximation and using a blend of BFS and DFS in order to find sets of disjoint augmenting paths up to a certain length, depending on the approximation parameter. He presented a randomized approximation scheme for the matching problem in general graphs: given a parameter $k \in \mathbb{N}$, with a small error probability it finds a $1 + 1/k$ approximation[1] using a number of passes independent

[1] If M is a matching and M^* is an optimal matching, then M is a $1+\varepsilon$ approximation if $|M^*| \leq (1+\varepsilon)\,|M|$. Sometimes, we specify approximation in percent: M being a ρ approximation with $\rho \in [0, 1]$ means that $|M| \geq \rho\,|M^*|$. For example, in our case, $k = 9$ means a 90% approximation.

of the input size. However, the dependence on the approximation parameter k is rather strong, namely $\Omega(k)^{\Omega(k)}$, even when restricting to the bipartite case. In [3], Eggert, Kliemann, and Srivastav gave a deterministic algorithm for the bipartite case requiring only $O(k^8)$ passes. Recently, Eggert, Kliemann, Munstermann, and Srivastav improved that to $O(k^5)$ passes [2]. The basic idea of all those approaches is to grow multiple disjoint alternating paths at the same time. In [2], as an edge e goes by in the stream, it is attempted to use it to extend any of the alternating paths constructed so far, provided certain conditions are met, either forming a longer alternating path or completing it to an augmenting one. Backtracking is used to "revive" paths that currently fail to grow any further. A scheme called *position limiting* is used to limit the ways the edges may be appended to the alternating paths; details will be explained later. Position limiting is important to establish the desired bound on the number of passes, but at the same time it is so designed that it does not get in the way of achieving the required approximation.

Main Results. We present a new approximation scheme for maximum matching in bipartite graph streams. The algorithm has a guaranteed approximation of $1 + 1/k$ (Thm. 1) and requires at most $O(kn)$ passes (Thm. 2). In principle, this theoretical bound is inferior to the $O(k^{O(1)})$-type bounds known for the previous algorithm, since it depends on the input size via n; but for smaller n this bound is better (e.g., for the $O(k^5)$ version, $n \leq 3 \times 10^6$ is "small" enough). A main contribution of this work is that experimentally our new algorithm outperforms the previously known one by two orders of magnitude, while already the latter stays far below its theoretical $O(k^5)$ bound. For example for $k = 9$, the previous algorithm exhibits a number of passes up to about 32 000 per instance. This is far below the theoretical $O(k^{O(1)})$-type bound, which is at least 14×10^6. However, the breakthrough lies in the pass requirement that we observe for our *new* algorithm: for $k = 9$ and up to $n = 2\,000\,000$ it never required more than 94 passes, with an appropriate choice of further parameters even no more than 65 passes. In our opinion, this work crosses the borderline between theory and practice towards really practically efficient streaming algorithms.

2 Our Algorithmic Innovation

Tree Structure. The basic challenge for streaming algorithms is: how to make the most out of a pass? When the task is to find augmenting paths, one usually maintains some structure incorporating alternating paths. Then the question more precisely reads: what kind of structure provides a high number of *extension points* where we can append new edges to it or which help to restructure it, as new information becomes available? In [3,2] we took a path-based approach. We grew multiple alternating paths in parallel, and the end of each path provided an extension point. In this new work, we present a tree-based approach. Trees are rooted at free vertices and each path starting at the root is alternating. *All vertices with an even distance to the root act as extension points.* This requires some further considerations: we have to adapt the position limiting scheme and

the stopping criterion. We prove the same approximation guarantee for the tree-based algorithm (Thm. 1) as we have for the path-based one. Regarding the number of passes, we only give a bound depending (linearly) on the size of the input; this is due to the new position limiting scheme. There is a simple work-around for this potential drawback: we can let the path-based algorithm run in parallel along the tree-based one, feeding both algorithms with the same edges as they go by in the stream. As soon as one of the algorithms terminates, we have a guaranteed approximation. This combined algorithm inherits the bound on the number of passes from the path-based one, which is independent of the input size. We did not implement this combination; given the experimental results this did not appear necessary.

Parameters. We make an extension that applies to both the path-based and tree-based algorithm. Our approximation technique is based on considering augmenting paths only up to a certain length, and to terminate when only a certain number of those remains. The length and the termination criterion both depend on the approximation parameter k. We introduce an additional parameter γ that is used to control a trade-off between path length and termination criterion. For the path-based algorithm, γ influences the worst-case bound on the number of passes: it ranges between $O(k^7)$ and $O(k^5)$. Experiments show that for some instance classes, the $O(k^7)$ version requires substantially fewer passes than the $O(k^5)$ one. This accentuates the importance of not only considering theoretical bounds. However, it should be noted that the path-based algorithm is by far outperformed by the tree-based one, independently of γ.

In addition to γ, another parameter $s \geq 1$, which we call the *stretch*, is introduced. It is related to the allowable length of the constructed augmenting paths and offers another trade-off control. For the path-based algorithm, only $s = 1$ makes sense, but larger values are meaningful for the tree-based one.

The following two sections, 3 and 4, describe the theoretical foundation of our approximation technique and explain the path-based algorithm from [2]. They also introduce the γ and s parameters. In Sect. 5, we present the new tree-based algorithm and in Sect. 6 the experimental setup and detailed discussion of results.

3 Approximation Technique

A *DAP algorithm* is one that finds, given a matching M, a set of disjoint augmenting paths. For $\lambda \in \mathbb{N}$, we call a path a λ *path* if it is of length at most $2\lambda+1$; the length of a path being the number of its edges. For $\lambda_1, \lambda_2 \in \mathbb{N}$, $\lambda_1 \leq \lambda_2$, a set $\mathcal{Y}$ of paths is called a (λ_1, λ_2) *DAP set* if:

1. All paths in $\mathcal{Y}$ are augmenting λ_2 paths.
2. Any two paths in $\mathcal{Y}$ are vertex-disjoint.
3. We cannot add another augmenting λ_1 path to $\mathcal{Y}$ without violating condition 2.

We call $s := \frac{\lambda_2}{\lambda_1}$ the *stretch*, since it specifies how far paths may stretch beyond λ_1. Given $\delta \in [0,1]$, a DAP algorithm is called a $(\lambda_1, \lambda_2, \delta)$ *DAP approximation algorithm* if it always delivers a result $\mathcal{A}$ of disjoint augmenting λ_2 paths such that there exists a (λ_1, λ_2) DAP set $\mathcal{Y}$ so that $|\mathcal{Y}| \leq |\mathcal{A}| + \delta\,|M|$. Let $\delta_{\text{inn}}, \delta_{\text{out}} \in [0,1]$ and DAP be a $(\lambda_1, \lambda_2, \delta_{\text{inn}})$ DAP approximation algorithm. All our algorithms utilize the loop shown in Algorithm 1. When this loop terminates, clearly there exists a (λ_1, λ_2) DAP set $\mathcal{Y}$ with $|\mathcal{Y}| \leq |\mathcal{A}| + \delta_{\text{inn}}\,|M| \leq \delta_{\text{out}}\,|M| + \delta_{\text{inn}}\,|M| = (\delta_{\text{inn}} + \delta_{\text{out}})\,|M|$, where M denotes the matching before the last augmentation. Let $k \in \mathbb{N}$ and $k \leq \lambda_1 \leq \lambda_2$ and

$$\delta(\lambda_1, \lambda_2) := \frac{\lambda_1 - k + 1}{2k\lambda_1\,(\lambda_2 + 2)} > 0 \ . \tag{1}$$

Algorithm 1. Outer Loop

1 $M :=$ any inclusion-maximal matching;
2 **repeat**
3 | $c := |M|$;
4 | $\mathcal{A} := \mathsf{DAP}(M)$;
5 | augment M using $\mathcal{A}$;
6 **until** $|\mathcal{A}| \leq \delta_{\text{out}}\, c$;

Following the pattern of [2, Lem. 4.1 and 4.2] we can prove:

Lemma 1. *Let M be an inclusion-maximal matching. Let $\mathcal{Y}$ be a (λ_1, λ_2) DAP set such that $|\mathcal{Y}| \leq 2\delta\,|M|$ with $\delta = \delta(\lambda_1, \lambda_2)$. Then M is a $1 + 1/k$ approximation.*

The lemma yields the $1 + 1/k$ approximation guarantee for Algorithm 1 when $\delta_{\text{inn}} = \delta_{\text{out}} = \delta(\lambda_1, \lambda_2)$. What are desirable values for λ_1 and λ_2? The DAP approximation algorithms presented in later sections (the path-based and the tree-based one) can work with any allowable setting for λ_1 and λ_2, so we have some freedom of choice. We assume that constructing longer paths is more expensive, so we would like to have those values small and in particular $\lambda_1 = \lambda_2$. (We will later encounter situations where it is conceivable that higher λ_2 is beneficial.) On the other hand, we would like to have δ large in order to terminate quickly. The function $\lambda \mapsto \delta(\lambda, \lambda)$ climbs until $\lambda = k - 1 + \sqrt{k^2 - 1} \leq 2k - 1$ and falls after that. Since we only use integral values for λ_1, the largest value to consider is $\lambda_1 = \lambda_2 = 2k - 1$. The smallest one is $\lambda_1 = \lambda_2 = k$. We parameterize the range in between by defining

$$\lambda(\gamma) := \lceil k\,(1 + \gamma) \rceil - 1 \quad \text{for each} \quad \gamma \in [1/k, 1] \ . \tag{2}$$

Consider the setting $\lambda_1 := \lambda_2 := \lambda(\gamma)$ and $\delta_{\text{inn}} := \delta_{\text{out}} := \delta(\lambda_1, \lambda_2)$. Then increasing γ increases path length, but also increases δ_{inn} and δ_{out}, which means that we are content with a less good approximation from the DAP algorithm and also relax the stopping condition of the outer loop. So γ controls a trade-off between path length and stopping criterion.

4 Path-Based DAP Approximation

We briefly describe how we find a $(\lambda_1, \lambda_2, \delta_{\text{inn}})$ DAP approximation with $\lambda_1 = \lambda_2$ in [2]; please consult that text for details. Fix an inclusion-maximal matching M. A vertex v is called *covered* if $v \in e \in M$ for some e, and *free* otherwise. We call an edge $e \in E$ a *matching edge* if $e \in M$, and *free* otherwise. Denote free(A) and free(B) the free vertices of partitions A and B, respectively. If $v \in V$ is not free, denote its *mate* by M_v, i.e., the unique vertex so that $\{v, M_v\} \in M$. We construct disjoint alternating paths starting at vertices of free(A), the *constructed paths*, and we index them by their starting vertices: $(P(\alpha))_{\alpha \in \text{free}(A)}$. When we find augmenting paths, they are stored in a set $\mathcal{A}$ and their vertices marked as *used*; a vertex not being used is called *remaining*. Denote remain(X) the remaining vertices in a set $X \subseteq V$. Suppose $P(\alpha) = (\alpha, e_1, b_1, m_1, a_1, \ldots, m_t, a_t)$ is a path with free vertex $\alpha \in \text{free}(A)$, vertices $a_1, \ldots, a_t \in A$ and $b_1, \ldots, b_t \in B$, free edges $e_1, \ldots, e_t \in E$ and matching edges $m_1, \ldots, m_t \in M$. Then we say that matching edge m_i has *position* i, $i \in [t]$. Each matching edge m has a *position limit* $\ell(m)$, initialized to $\ell(m) := \lambda_1 + 1$. We perform *position limiting*, i.e., a matching edge m will only be inserted into a constructed path if its new position is strictly smaller than its position limit. When a matching edge is inserted, its position limit is decremented to its position in the constructed path.

After each pass, we backtrack conditionally: each constructed path that was not modified during that preceding pass has its last two edges removed. When the number of constructed paths of positive length falls on or below $\delta_{\text{inn}} |M|$, we terminate and deliver all augmenting paths found so far. Position limiting is important for proving the bound $2\lambda_1 \delta_{\text{inn}}^{-1} + 1$ on the number of passes of this DAP algorithm [2, Lem. 7.1]. By the stopping criterion of the outer loop, it is invoked at most $\delta_{\text{out}}^{-1} + 1$ times [2, Thm. 7.2]. Hence, with (2), we have the following bound on the number of passes conducted in total: $(\delta_{\text{out}}^{-1} + 1)\ (2\lambda_1 \delta_{\text{inn}}^{-1} + 1) = O(\gamma^{-2}\, k^5)$. Let us specify γ by $\tilde{\gamma} \in [0, 1]$ via the relation $\gamma = k^{-\tilde{\gamma}}$. Then for $\tilde{\gamma} = 0$ the bound is $O(k^5)$, for $\tilde{\gamma} = 1/2$ it is $O(k^6)$, and for $\tilde{\gamma} = 1$ it is $O(k^7)$. We compare these three values for $\tilde{\gamma}$ in experiments.

5 Tree-Based DAP Approximation

An *alternating tree* is a pair consisting of a tree T that is a subgraph of G, and a vertex $r \in V(T)$, called its *root*, so that each path from r to any other vertex of T is an alternating path. For $v \in V(T)$ the subtree induced by all vertices reachable from r via v is called the *subtree below* v and denoted $T[v]$. An *alternating forest* consists of one or more alternating trees being pairwise vertex-disjoint. Our tree-based DAP algorithm maintains an alternating forest with trees indexed by vertices. We write $T(v) = (V(v), E(v))$ for the tree indexed with $v \in V$; we ensure that it is either empty or rooted at v. The *forest* $\mathcal{F}$ is $\mathcal{F} = \{T(v);\ v \in \text{remain}(V)\}$. We only deal with trees from $\mathcal{F}$. We call a tree *properly rooted* if its root is a free vertex. A properly rooted tree $T(\alpha)$ together with an edge $\{a, \beta\}$ with β being free and $a \in V(T)$ at an even distance from α, yield an augmenting path.

We initialize by setting $T(\alpha) \coloneqq (\{\alpha\}, \emptyset)$ for each $\alpha \in \text{free}(A)$ and $T(v) \coloneqq (\emptyset, \emptyset)$ for each $v \in V \setminus \text{free}(A)$. So we have empty trees and one-vertex trees with a free vertex of A. Position limits are initialized $\ell(m) \coloneqq \lambda_1 + 1$ for each $m \in M$ as usual. If $(\alpha, e_1, b_1, m_1, a_1, \ldots, m_t, a_t)$ is a path in the properly rooted tree $T(\alpha)$, then we say that matching edge m_i, $i \in [t]$, has *position* i. Results (i.e., the augmenting paths found) will be stored into a set $\mathcal{A}$, that is initialized to $\mathcal{A} \coloneqq \emptyset$.

Trees grow over time, and there may also emerge non-properly rooted trees. When a free edge $\{a, b\}$ between two remaining vertices goes by in the stream with b being covered, the algorithm checks whether to extend any of the trees. Conditions are: the tree has to be properly rooted, say $T(\alpha)$, it must contain a, and $i < \ell(\{b, M_b\})$, where i is the position that the matching edge $\{b, M_b\}$ would take in $T(\alpha)$. If all those conditions are met, an *extension step* occurs: the two edges $\{a, b\}$ and $\{b, M_b\}$ are added to $T(\alpha)$, and, if $\{b, M_b\}$ is already part of a tree $T(b')$, then $T(b')[b]$ is removed from $T(b')$ and connected to $T(\alpha)$ via $\{a, b\}$. The tree $T(b')$ is not required to be properly rooted, but it may be. Bipartiteness ensures that $M_b \in V(T(b')[b])$. Position limits for all inserted or migrated edges are updated to reflect their new positions.

When a free edge $\{a, \beta\}$ with $a, \beta \in \text{remain}(V)$ goes by in the stream with β being *free*, then we check whether we can build an augmenting path. If there is a properly rooted tree $T(\alpha)$ with $a \in V(\alpha)$, the path P in $T(\alpha)$ from α to β is augmenting. In that case, a *completion step* occurs: we store P into the result set $\mathcal{A}$, and mark all vertices on P as used. Also, we adjust our forest as follows. For each $a \in V(P) \cap A$ and each of its neighbors in $T(\alpha)$ and not in P, i.e., for each $b \in N_{T(\alpha)}(a) \setminus V(P)$, we set $T(b) \coloneqq T(\alpha)[b]$. In other words, we "cut" P out of $T(\alpha)$ and make each of the resulting subtrees that "fall off" a new tree of its own. None of those is properly rooted, and also they are rooted at vertices of partition B, not A as the properly rooted ones. However, they – or parts of them – can subsequently be connected to remaining properly rooted trees by an extension step as described above. One last and crucial feature of the completion step is *position limit release*: we release position limits to $\lambda_1 + 1$ on edges of the new (non-properly rooted) trees. This is important for the proof of the approximation guarantee in Lem. 2. We do not explicitly backtrack; yet, position limit release can be considered a form of backtracking.

Position limit release requires further considerations. In an extension step, although position limits at first are not higher than $\lambda_1 + 1$, edges can be included in a tree at positions beyond λ_1. Assume $m = \{b, M_b\}$ is inserted at position $i < \lambda_1$ into a properly rooted tree $T(\alpha)$ and subsequently, more edges are inserted behind m. Then an augmenting path is found in $T(\alpha)$ not incorporating m, hence the position limit of m is released. Later m can be inserted at a position j with $i < j \leq \lambda_1$ in another properly rooted tree $T(\alpha')$. When m carries a sufficiently deep subtree with it, then $T(\alpha')$ could grow beyond λ_1, even though $j \leq \lambda_1$. Here, the second length parameter λ_2 comes into play. When the migrated subtree is too deep, we trim its branches just so that it can be migrated without making the destination tree reach beyond λ_2. The trimmed-off branches become

non-properly rooted trees of their own. We control a trade-off this way: higher λ_2 means fewer trimming and hence fewer destruction of previously built structure. But higher λ_2 reduces $\delta(\lambda_1, \lambda_2)$ and so may prolong termination. Choosing $\lambda_2 := \lambda_1$ is possible.

After each pass, it is checked whether it is time to terminate and return the result $\mathcal{A}$. We terminate when any of the following two conditions is met:

(T1) During the last pass, no extension or completion occurred. In other words, the forest did not change. (It then would not change during further passes.)
(T2) The number of properly rooted trees (which is also the number of remaining free vertices of A) is on or below $\delta_{\text{inn}} |M|$.

Lemma 2. *The algorithm described in this section is a* $(\lambda_1, \lambda_2, \delta_{\text{inn}})$ *DAP approximation algorithm.*

Proof (Sketch). When the algorithm terminates via condition (T2), it could have, by carrying on, found at most $\delta_{\text{inn}} |M|$ additional augmenting paths. We show that when we restrict to termination condition (T1), we have a $(\lambda_1, \lambda_2, 0)$ DAP approximation algorithm. To this end, it only remains to show that we cannot add an augmenting λ_1 path to $\mathcal{A}$ without hitting at least one of the paths already included. Suppose there is an augmenting path $(\alpha, e_1, b_1, m_1, a_1, e_2, b_2, m_2, a_2, \ldots, a_t, e_{t+1}, \beta)$ with $t \leq \lambda_1$, $\alpha \in \text{free}(A)$ and $\beta \in \text{free}(B)$ that is disjoint to all paths in $\mathcal{A}$. We can show by induction that when the algorithm terminates, then a_t is in a properly rooted tree T; this part of the proof depends crucially on position limit release. This claim helps establishing a contradiction: first, by the stopping criterion, a_t was there for the whole pass, since an extension step would have made termination impossible. But then the algorithm would have pulled out an augmenting path from T when $e_{t+1} = \{a_t, \beta\}$ came along in the stream during the last pass and so it would not have been allowed to terminate. □

Following from Lem. 1 and 2 we have:

Theorem 1. *Algorithm 1 with the tree-based DAP approximation algorithm described in this section gives a* $1 + 1/k$ *approximation, if* $\delta_{\text{inn}} = \delta_{\text{out}} = \delta(\lambda_1, \lambda_2)$.

As for the number of passes, at this time we only have a bound depending on the problem size. The main hindrance for an independent bound is position limit release, as can be seen by a comparison with the techniques in [2, Lem. 7.1].

Theorem 2. *Algorithm 1 with the tree-based DAP approximation algorithm described in this section requires at most* $\frac{\lambda_1 n}{4} + 1 \leq \frac{kn}{2} + 1$ *passes.*

Proof (Sketch). Consider an invocation of the DAP algorithm for a matching M. For each $m \in M$ denote $\text{dist}(m)$ the minimum position of m over all alternating paths that start at a free vertex of A and use only remaining vertices (regardless whether the path can be realized by the algorithm as part of a tree). If there is no such path, we put $\text{dist}(m) := \infty$. All matching edges m that are eligible to be inserted into a (properly rooted) tree of our forest have $\text{dist}(m) \leq \lambda_1$. It can be seen easily by induction that while no further augmenting path is found,

Table 1. Results for path-based (indicated by "p" in the left column) and tree-based ("t") algorithms. Parameters are $\lambda_1 = \lambda(\gamma)$ as per (2), $\gamma = k^{-\tilde{\gamma}}$, and stretch $s = \frac{\lambda_2}{\lambda_1}$. Numbers state the maximum and rounded mean number of passes, respectively, that were observed for the different choices of parameters and instance classes. We have relatively small instances: $n = 40\,000, 41\,000, \dots, 50\,000$. Density is limited by $D_{\max} = 1/10$. Number of edges ranges up to about $|E| = 62 \times 10^6$.

	$\tilde{\gamma}$	s	maximum					mean				
			rand	degm	hilo	rbg	rope	rand	degm	hilo	rbg	rope
p	0	1	11 886	14 180	7 032	4 723	2 689	107	145	3 337	257	378
p	1/2	1	7 817	31 491	7 971	4 383	3 843	80	127	2 071	500	541
p	1	1	7 121	32 844	9 106	5 687	5 126	74	166	2 033	844	790
t	0	1	6	9	75	41	79	3	3	51	5	22
t	0	2	6	9	74	52	94	3	3	51	5	26
t	1/2	1	6	9	59	37	63	3	3	38	5	20
t	1/2	2	6	9	59	44	70	3	3	38	5	22
t	1	1	6	9	54	38	61	3	3	35	5	20
t	1	2	6	9	55	40	67	3	3	36	6	21

after at most λ_1 passes, for each m with $\text{dist}(m) \leq \lambda_1$, we have $\ell(m) = \text{dist}(m)$. Thus, after at most λ_1 passes, a new augmenting path is found or the algorithm terminates due to lack of action, i.e., by termination condition (T1). Since the initial matching is already a 50% approximation, the former can happen at most $\frac{|M^*|}{2} \leq \frac{n}{4}$ times over *all* executions of the DAP algorithm, with M^* denoting an optimal matching. □

6 Experiments

Setup. We use randomly generated instances with various structure:

rand: Random bipartite graph; each edge occurs with probability $p \in [0, 1]$.

degm: The degrees in one partition are a linear function of the vertex index. A parameter $p \in [0, 1]$ is used to scale degrees.

hilo, rbg, rope: Vertices in both partitions are divided into l groups of equal size and connected based on that according to different schemes. For details on hilo and rbg (the latter also known as fewg or manyg), we refer to [9,1,6]. In rope [9,1], the construction results in a layered graph, where the odd layers are perfect matchings, and the even layers are random bipartite graphs with parameter $p \in [0, 1]$. Such a graph has a unique perfect matching.

Instances are kept completely in RAM, so the streaming situation is only simulated. For Tab. 2, we impose a hard limit of 1×10^9 edges, meaning about 7.5 GiB (each vertex is stored as a 32 bit unsigned integer). A *series* is specified by a density limit and a set of values for n. For each n of a series and for each class, we generate 256 instances on n vertices. For hilo, rbg, and rope, parameter l is chosen randomly from the set of divisors of $|A| = n/2$. For all classes, a parameter controlling the (expected) number of edges (e.g., p for rand) is being moved

Table 2. Using trees and a good setting determined in previous series, namely $\tilde{\gamma} = 1$ and $s = 1$. We treat larger instances: $n = 100\,000, 200\,000, \ldots, 1\,000\,000$. Density is limited by $D_{\max} = 1/10$. Development for growing n is shown. Number of edges ranges up to about $|E| = 1 \times 10^9$, which takes about 7.5 GiB of space.

	maximum					mean				
n	rand	degm	hilo	rbg	rope	rand	degm	hilo	rbg	rope
100 000	3	8	53	30	62	2.5	3.2	35.0	5.1	19.8
200 000	3	7	56	31	63	2.5	2.8	37.6	4.7	19.1
300 000	3	7	55	29	64	2.5	2.9	38.6	3.9	18.2
400 000	3	8	56	33	63	2.5	2.9	36.3	5.3	15.6
500 000	3	7	58	34	64	2.5	3.0	36.7	4.4	19.4
600 000	3	9	58	30	64	2.5	3.5	38.4	3.3	18.1
700 000	6	9	56	35	62	2.5	3.6	37.4	3.9	18.5
800 000	3	8	58	31	63	2.5	3.5	37.9	3.1	16.2
900 000	7	8	61	32	62	2.6	3.3	37.0	3.7	14.5
1 000 000	6	9	60	34	65	2.5	3.1	33.4	4.6	18.2

through a range such that we start with very few (expected) edges and go up to (or close to) the maximum number of edges possible, given the hard limit, the limit $D_{\max}$ on the density (allowing some overstepping due to randomness), and any limit resulting from structural properties (e.g., number of groups l). This way we produce instances of different densities. For rand and degm, we use 16 different densities and generate 16 instances each. For hilo, rbg, and rope, we use 64 random choices of l and for each 4 different densities. This amounts to 256 instances per n and class. In total, we treated more than 60 000 instances. After an instance is generated, its edges are brought into random order. Then each algorithm is run on it once, and then again with partitions A and B swapped. During one run of an algorithm, the order of edges in the stream is kept fix.

Results. Result details are given in Tables 1, 2, and 3. We only consider numbers of passes and completely neglect running times. The approximation parameter is fixed to $k = 9$, which means a guaranteed 90% approximation. We limit the graph density $D = \frac{|E|}{|A|\,|B|}$ to $1/10$ or lower, since preliminary tests had shown that usually the tree-based algorithm already exhibits its worst number of passes there. The density limit saves computation time, since each single pass takes fewer time when there are fewer edges. We give maximum and mean numbers of passes, while we focus the discussion on the maximum.

Table 1 compares the path-based and tree-based algorithms. The tree-based outperforms the path-based by a large margin, in the worst and the average case. For no setting of $\tilde{\gamma}$ or s, the tree-based algorithm needed more than 94 passes, whereas the path-based one ranges up to more than 32 000 passes. This also means that the main improvement stems from using trees instead of paths, and not from the trade-off parameters. For the path-based algorithm, there are considerable differences for different values of $\tilde{\gamma}$. There is no best setting for $\tilde{\gamma}$, but it depends on the instance class: compare in particular the rand and rbg or

Table 3. The same algorithm and parameters as in Tab. 2, but larger number of vertices and lower density limit, namely $D_{\max} = 1 \times 10^{-4}$. Development for growing n is shown. Number of edges ranges up to about $|E| = 100 \times 10^6$.

	maximum					mean				
n	rand	degm	hilo	rbg	rope	rand	degm	hilo	rbg	rope
1 000 000	48	43	62	41	48	5.5	8.8	29.6	4.5	17.7
1 100 000	47	49	60	42	50	5.1	8.6	29.6	4.4	17.0
1 200 000	45	51	60	33	52	4.4	8.4	29.0	5.2	14.5
1 300 000	31	30	59	41	47	3.9	8.7	30.0	3.9	15.5
1 400 000	32	35	61	35	51	4.5	7.6	28.5	4.6	14.9
1 500 000	28	29	57	33	51	3.9	8.5	28.7	4.5	15.5
1 600 000	25	27	58	34	52	4.1	6.9	26.7	4.5	15.9
1 700 000	28	42	60	35	52	3.6	7.7	28.8	4.6	16.2
1 800 000	31	29	60	35	54	4.1	6.8	28.1	3.2	15.2
1 900 000	23	26	56	34	50	3.2	6.3	27.7	4.6	14.0
2 000 000	32	21	60	35	49	3.4	6.4	28.9	4.5	15.7

rope classes. For the hilo class, the maximum and the mean number of passes move in opposite directions when changing $\tilde{\gamma}$. For the tree-based algorithm, $\tilde{\gamma} = 1$ and $s = 1$ is a good setting, with maximum number of passes not exceeding 61. For the tree-based algorithm, the highest number of passes is attained for $\tilde{\gamma} = 0$, namely 79 for $s = 1$ and 94 for $s = 2$.

Tables 2 and 3 consider the tree-based algorithm for larger instances. In particular they address the concern that the maximum number of passes could grow with the number n of vertices. There is a slight upward tendency for some classes, e.g., for hilo in Tab. 2. The maximum number of passes in the range $n = 100\,000$ to $n = 1\,000\,000$ is attained for each n by rope in Tab. 2. The smallest value is 62 and the largest 65. From this perspective, the increase is below 5% while the number of vertices grows by factor 10. For even larger n and smaller density, Tab. 3, the maximum is 62. Additionally, we conducted two series with density limits 1×10^{-3} and 1×10^{-4}, respectively, and up to $n = 1 \times 10^6$. The maximum observed in those series was 60.

7 Conclusion and Future Work

The tree-based algorithm outperforms the path-based one by a large margin. Its theoretical pass bound depends on n, but experiments give rise to the hypothesis that there in fact is at most a minor dependence. Future work will put this hypothesis to a test, in particular we will consider larger instances.

As an algorithmic addition, one could use multiple matchings in parallel. This follows the same guiding question: how to make the most out of each pass? We initialize a number of matchings in a randomized manner. Then we work on all of those at once: when an edge comes along in the stream, it might not be usable to extend all of the forests, but maybe some of them.

Concerning running time, we have preliminary results that in certain cases, our tree-based algorithm outperforms random-access algorithms – even if the instance completely fits into random-access memory. This may be due to streaming algorithms making better use of memory caches. This will be addressed in detail in future work.

Acknowledgments

I thank Ole Kliemann for helpful discussions, supporting the experiments, and for profiling and improving our C++ implementation. I thank Anand Srivastav and the anonymous referees for helpful comments on my manuscript. I thank the Rechenzentrum at Universität Kiel for many months of computation time. I thank the Deutsche Forschungsgemeinschaft (DFG) for financial support through Priority Program 1307, *Algorithm Engineering*, Grant Sr7/12-2.

References

1. Cherkassky, B.V., Goldberg, A.V., Martin, P.: Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. ACM Journal of Experimental Algorithms 3 (1998), http://www.jea.acm.org/1998/CherkasskyAugment/
2. Eggert, S., Kliemann, L., Munstermann, P., Srivastav, A.: Bipartite matching in the semi-streaming model. Tech. Rep. 1101, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, document ID: 519a88bb-5f5a-409d-8293-13cd80a66b36 (2011), http://www.informatik.uni-kiel.de/~lki/tr_1101.pdf
3. Eggert, S., Kliemann, L., Srivastav, A.: Bipartite graph matchings in the semi-streaming model. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 492–503. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04128-0_44, available also at http://www.informatik.uni-kiel.de/~discopt/person/asr/publication/streaming_esa_09.pdf, Presented also at the MADALGO Workshop on Massive Data Algorithmics, Århus, Denmark (June 2009)
4. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 531–543. Springer, Heidelberg (2004), http://dx.doi.org/10.1007/b99859
5. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM Journal on Computing 2(4), 225–231 (1973)
6. Langguth, J., Manne, F., Sanders, P.: Heuristic initialization for bipartite matching problems. ACM Journal of Experimental Algorithmics 15, 1.3:1.1–1.3:1.22 (2010), http://doi.acm.org/10.1145/1712655.1712656
7. McGregor, A.: Finding graph matchings in data streams. In: Chekuri, C., Jansen, K., Rolim, J.D.P., Trevisan, L. (eds.) APPROX 2005 and RANDOM 2005. LNCS, vol. 3624, pp. 170–181. Springer, Heidelberg (2005), http://dx.doi.org/10.1007/11538462_15

8. Muthukrishnan, S.M.: Data streams: Algorithms and applications. Foundations and Trends in Theoretical Computer Science 1(2), 67 pages (2005), http://algo.research.googlepages.com/eight.ps
9. Setubal, J.C.: Sequential and parallel experimental results with bipartite matching algorithms. Tech. Rep. IC-96-09, Institute of Computing, University of Campinas, Brazil (1996), http://www.dcc.unicamp.br/ic-tr-ftp/1996/96-09.ps.gz

Beyond Unit Propagation in SAT Solving*

Michael Kaufmann and Stephan Kottler

Wilhelm–Schickard–Institut für Informatik, University of Tübingen, Germany

Abstract. The tremendous improvement in SAT solving has made SAT solvers a core engine for many real world applications. Though still being a branch-and-bound approach purposive engineering of the original algorithm has enhanced state-of-the-art solvers to tackle huge and difficult SAT instances. The bulk of solving time is spent on iteratively propagating variable assignments that are implied by decisions.

In this paper we present two approaches on how to extend the broadly applied Unit Propagation technique where a variable assignment is implied iff a clause has all but one of its literals assigned to *false*. We propose efficient ways to utilize more reasoning in the main component of current SAT solvers so as to be less dependent on felicitous branching decisions.

1 Introduction

Satisfiability checking (SAT) is one of the well known NP-hard problems [15] in both theoretical and practical computer science. There are several real-world applications that are actually tackled by modelling parts of these problems as SAT instances like hardware and software verification [34], planning [23], and bioinformatics [28].

The collaboration of theoretical research and algorithm engineering has managed to design and realise highly optimised SAT solvers. Different types of solvers have been engineered for different types of SAT instances (i.e. real-world, random, handmade). Solvers for real-world applications which are the objective of the presented work compensate the little effort spent for decision making by very fast search and propagation of decisions [12]. Though this often works for a wide range of instances solvers are highly sensitive to an initial random seed (affecting about 1-2% of decisions [19,10]) and minor changes of parameters.

The fundamental *Unit Propagation* technique where a variable assignment is implied iff a clause has all but one of its literals assigned to *false* has now been highly optimised and nearly exploited for further speed-ups, so solver engineering has recently taken the line to improve at other stages of solving, e.g. to emphasise more resolution [9,11] and enhanced analysis of conflicting assignments [3,20,31].

However, with this work we are heading for further improvements of propagation from a different point of view. Our motivation is to increase the number of

* This work was supported by DFG-SPP 1307, project “Structure-based Algorithm Engineering for SAT-Solving”.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 267–279, 2011.
© Springer-Verlag Berlin Heidelberg 2011

implications that follow from one decision. As shown in [35,26] many industrial SAT instances may only require a very small set of variables that are chosen as decision variables. The difficulty is to find such a set of variables. However, an increase of the average number of implications that are caused by one decision will go along with an even smaller set of variables that have to be chosen as decision variables. Moreover, a smaller number of decisions and a greater number of implications reduces the dependency on felicitous branching decisions.

Our main idea is to consider a clause C for propagation even if it has more than one literal unassigned. The set of unassigned literals in clause C might have a common implication l_i. If so, l_i can be propagated even though C is not unit. In this process we use all binary clauses of a formula to check for common implications of literals. In the next section we give basic definitions and sketch the state-of-the-art of SAT solving. Section 3 explains two approaches on how to realise extended propagation. In section 4 the two approaches are evaluated. Section 5 finally concludes this work.

2 Basic Definitions and State of the Art

Practical SAT solving is mostly about Boolean formulae in *conjunctive normal form* (CNF). A formula F in CNF consists of a set of Boolean variables $\mathcal{V}_F$ and a set of clauses $\mathcal{C}_F$ that are connected as conjunctions. A clause $C \in \mathcal{C}_F$ is a disjunction of $|C|$ literals, whereas each literal l is either a variable $v \in \mathcal{V}_F$ or its negation ($l = \overline{v}$). A *partial assignment* $\tau_V : V \mapsto \{false, true\}$ is a function that assigns Boolean values to all variables of $V \subseteq \mathcal{V}_F$. If $V = \mathcal{V}_F$ then τ_V is said to be a *complete assignment* for F. A formula F is *Satisfiable* iff there exists an assignment τ_V with $V \subseteq \mathcal{V}_F$ that fulfills all clauses in $\mathcal{C}_F$. A clause is fulfilled if at least one of its literals is true. A literal l of variable v is true if $l = v$ and $\tau_V(v) = true$ or if $l = \overline{v}$ and $\tau_V(v) = false$. We write $\tau_V(l) = false/true$ for short. A clause C is unit if $|C| = 1$ and binary if $|C| = 2$.

SAT solvers can generally be categorized into two distinct types, namely *complete* and *incomplete* solvers. Given a formula F in CNF both kinds of solvers may compute a satisfying assignment for F. However, complete solvers can also prove *Unsatisfiability* for formulae that cannot be satisfied by any assignment to the variables in $\mathcal{V}_F$.

Incomplete solvers are mostly local search approaches [32] which have shown to be especially successful for satisfiable random SAT instances [1]. Complete solvers are based on the DPLL algorithm [18,17] that may be classified as branch and bound algorithm. State-of-the-art solvers are predominantly variants of *conflict driven solving with clause learning* (CDCL) that is an extension of the original DPLL algorithm [29]. Algorithm 1 sketches the CDCL approach.

For simplicity we assume the formula F to be simplified in the way that no unit clauses are contained. As long as there are still unassigned variables (line 5) a branching choice is made in line 6. More details about the common branching heuristic 'VSIDS' can be found in [30,19]. In the following line 7 all implications (i.e. implied variable assignments) of the chosen decision literal are computed by the so-called *Boolean Constraint Propagation* (BCP). In subsection 2.1 BCP will

Algorithm 1. Sketch of the CDCL approach

```
1  Require Formula F in CNF without unit clauses ;
2  Function CDCL(F)
3      A ← ∅ /* τ_A is current partial assignment                    */
4      V_U ← vars(F) /* set of unassigned variables                   */
5      while V_U ≠ ∅ do
6          l ← choose-next-decision(V_U) ;
7          A' ← BCP(l) ;
8          if A' in conflict with A then
9              L ← analyze-conflict(A, A') ;
10             if L = ∅ then return 'Unsatisfiable';
11             F ← F ∪ L;
12             A ← backjump(L) ;
13         else
14             A ← A ∪ A';
15         V_U ← V_U \ {vars(A)};
16     return 'Satisfiable' /* assignment A satisfies F              */
```

be explained in more detail. If an assignment to a variable $w \in \mathcal{V}$ is implied that contradicts its current state (e.g. w is already $b \in \{false, true\}$ but is implied to be $\overline{b}$) a *conflict* is found (line 8).

If there is no conflict the partial assignment can be adapted (line 14) and the loop continues. Otherwise the function *analyze-conflict* computes a new clause L that expresses the current conflict as a single condition. L contains only literals which are $false$ by the current partial assignment A and the implications of decision l. If L is empty the formula F is unsatisfiable. Otherwise L will be (temporarily) added to F. The solver jumps back to a previous partial assignment such that (at least) one literal in L becomes unassigned. More details about *clause learning* and *backjumping* can be found in [30,36,7]. If the main loop terminates an assignment is found that satisfies all clauses in F.

2.1 Boolean Constraint Propagation

The main ingredient of any conflict driven SAT solver is clearly a fast implementation of Boolean Constraint Propagation (BCP). For the bulk of SAT instances more than 80% of the runtime is spent on Unit Propagation [30,25]. On the other hand relatively little computational effort is spent on choosing decision variables. This constitutes ongoing research to improve the speed of BCP [30,10,14].

For CDCL based SAT solvers BCP exactly corresponds to *Unit Propagation*. If all but one literals of a clause $C_j \in \mathcal{C}_F$ are false under an assignment τ_V (the clause is *unit* under τ_V) the remaining literal has to become true. More precisely: Let $C_j \in \mathcal{C} = \{l_1, \vee l_2 \ldots \vee l_k\}$ and $\tau_V(l_i) = false \; \forall \; 2 \leq i \leq k$. Thus C_j is unit under τ_V and $\tau_V(l_1) \leftarrow true$ is implied. Unit propagation applies this rule until no more implications can be derived.

In order to analyse a conflict (line 9 of Algorithm 1) a CDCL solver maintains a so-called *reason* for each assigned variable. This reason can either be empty for decision variables or it stores the clause which forced the variable to be assigned by Unit Propagation. In the example the variable of l_1 has reason C_j.

2.2 Related Work

The application of more and advanced reasoning in SAT solving has been studied in several different contexts [33,5]. The so-called *Look-ahead solvers* [27,8,21,12] aim to improve the quality of branching decisions and thus to guide the solvers search. One of the basic ideas is to propagate both assignments of a variable x before a decision on x is finally made. In doing so further reductions may be applied to the formula [27,8,22]. Look-ahead solvers are particularly successful for handmade and unsatisfiable random instances[1]. Our approach differs in this respect that Unit Propagation itself is extended rather than the decision process.

Recent work [3,31,20] improves on the quality of clause learning. In [31] the authors focus on particular *missed implications* that are not caught by Unit Propagation. Conflict analysis is modified to learn some additional clauses in order to improve the quality of subsequent propagation. Our approach aims to tackle the missed implications already at each propagation step.

Our extension of Unit Propagation is eminently based on the set of binary clauses in a formula. In [6,4,13,33] special treatments of binary clauses are proposed. Bacchus introduces the concept of *Hyper-binary resolution* in combination with the computation of the complete binary closure. Applying this approach at each level of the SAT search captures the implications generated with our approach. However, this turns out to be too time consuming for today's SAT instances. In [13], a similar idea was used but only as a preprocessor for SAT formulae.

3 Enhancing BCP

In this work we aim to improve Boolean Constraint Propagation (BCP) in terms of the number of implications that can be derived from one decision. This is contrary to a common focus in research on how to improve the speed of Unit Propagation [30,14]. In this section we introduce two ideas on how to extend classical BCP. An evaluation and comparison of both implementations are given in section 4. We use basic concepts from graph theory and algorithms e.g. [16].

3.1 General Observations on Clauses and Implications

In classical Unit Propagation any clause $C_j \in \mathcal{C}$ can imply the value for at most one variable. As described above this applies if all but one literals of C_j are falsified by a partial assignment of the variables. However, it may happen that C_j does not have to become unit until the value of a variable can be implied. This goes along with the fact that C_j may directly imply values for more than one variable.

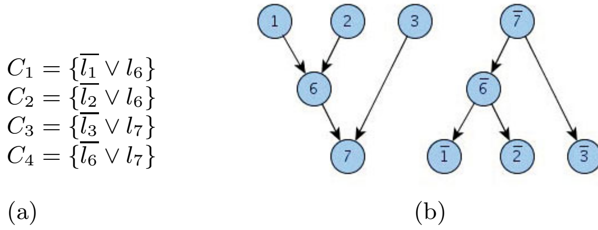


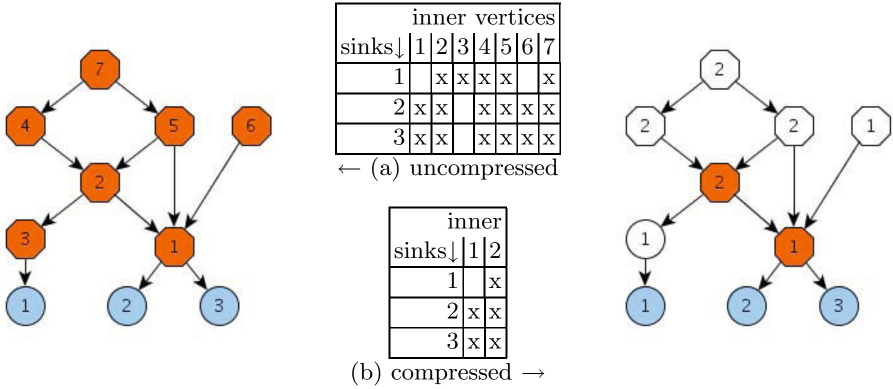
Fig. 1. Implication graph induced by binary clauses

Consider the following example: Given a clause $C_5 = \{l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5\}$ and a partial assignment τ_V such that $\tau_V(l_5) = \tau_V(l_4) = false$. Apparently Unit Propagation can not be applied for clause C_5 since there is more than one literal unassigned in C_5. Assume there are also the binary clauses shown in Figure 1 (a). Note that binary clauses constitute a special constraint in CNF: Any binary clause can be understood as two implications. By applying the idea of Unit Propagation one can claim that if one of the two literals is false the value of the other variable is implied. This constitutes an implication graph as this is used in [2] to solve 2-CNF formulae. Each variable $v_i \in \mathcal{V}$ of the formula is represented by two vertices $l_i, \overline{l_i}$ in the graph, one for the state of v being *true* (l_i) and the second for the state of v being *false* ($\overline{l_i}$). Each binary clause $\{l_i \vee l_j\}$ causes two directed edges $(\overline{l_i} \rightarrow l_j), (\overline{l_j} \rightarrow l_i)$ as shown in Figure 1 (b). This kind of graph was also used in [6] to compute Hyper-Binary Resolution in a SAT preprocessor. Examine clause C_5 again. We know that one of the literals l_1, l_2 or l_3 has to fulfill C_5 since the others are *false* under τ_V. Considering the implication graph all three literals l_1, l_2, l_3 imply literal l_7 since there is a chain of implications (a path) from all three to l_7. Detecting such cases allows for further implications that are beyond Unit Propagation. However, BCP is a highly critical part in CDCL solving and therefore requires fast execution. In the next two subsections we present two ideas on how to realise the extended propagation more or less efficiently.

3.2 A Matrix Based Approach

To extend Boolean Constraint Propagation as described above the algorithm has to determine if for a given clause all unassigned variables have some common implication. This is equal to a reachability problem in the implication graph. Additional computation like breadth–first search during BCP is beyond question. Clearly, an adjacency matrix of vertices (i.e. literals) could allow for fastest computation. However, industrial SAT instances range up to 10 million variables which makes a quadratic matrix evidently infeasible. To cope with a high number of literals by allowing random matrix access we formulate the following observation:

Observation 1. *Given a directed acyclic graph $G = (V, E)$, two vertices $a, b \in V$ reach a common vertex iff there exists at least one sink $s \in V$ and two paths $(a \rightharpoonup s), (b \rightharpoonup s)$. A sink is any vertex without outgoing edges.*



sinks↓ \ inner vertices	1	2	3	4	5	6	7
1		x	x	x	x		x
2	x	x		x	x	x	x
3	x	x		x	x	x	x

← (a) uncompressed

sinks↓ \ inner	1	2
1		x
2	x	x
3	x	x

(b) compressed →

Fig. 2. Sink matrix of one component. Inner vertices are drawn as dark octagons, sinks are drawn bright ellipses. In (b) the compression of the matrix by reduction of epigone vertices is given. Epigone vertices have transparent fill colours.

We denote the set of all sinks in G by $\sigma \subseteq V$. With Observation 1 reachability information is only required for pairs of vertices $a, b \in V$ where one vertex is a sink ($b \in \sigma$) and the other is an inner vertex ($a \in V \setminus \sigma$). Although Observation 1 is evident this already reduces the size of a reachability matrix drastically in practice (cf. Section 4). Moreover, in our application the implication graph often decomposes into several (disconnected) components $\gamma_j \in V$. Hence, the functionality of the adjacency matrix can be achieved by holding several independent $n_j \times s_j$ - matrices, with j being the index, s_j the number of sinks and n_j the number of inner vertices of the j-th component γ_j. An example of a reachability matrix is given in Figure 2(a). Note that the indices of sinks and inner nodes are consistently assigned within each component. The observation requires an acyclic directed graph. The removal of strongly connected components (SCC) can be combined with *equivalence reasoning*. Literals belonging to the same SCC are identical and can be replaced by one representing literal as in [6,9]. Thus, by computing SCCs the algorithm detects equivalent literals and furthermore, achieves the requirements of Observation 1.

First and foremost the aim of the matrices is to predict whether or not a set $V' \subset V$ of vertices reaches a common vertex. It is not important at first place to exactly know the sink that can be reached when starting from V'. Moreover, the matrices are only consulted to provide answers into one direction: "Do some vertices have a common successor?", but not the other way around: "Is a sink reached by some particular inner vertices?". This allows for further lossless compression of a reachability matrix by Observation 2.

Observation 2. *Let $N(v)$ be the adjacent vertices of $v \in V$. If any inner vertex $v \in V \setminus \sigma$ reaches exactly the same sinks as one of its successors $w \in N(v)$ then v can adopt the reachability information of w. We call v an* epigone *of w.*

In particular with Observation 2 any epigone v of w can be reduced: if w is a sink then v becomes a sink with the same sink ID as w. Otherwise v becomes an

inner vertex sharing the same column as w in the reachability matrix. In Figure 2 the compression of a reachability matrix is shown in an example.

The reachability matrices can further be compressed by omitting leading and ending blank entries in each column. Two additional integers can give the range for each column. We only apply this for a column if the overhead of the additional integers does not exceed the saving of memory.

Maintenance of reachability matrices. Before the initialisation of the solver the vertex of each literal l_i is assigned to its own component $\gamma(l_i)$. Whenever a binary clause $\{l_i \vee l_j\}$ is added the two components $\gamma(\overline{l_i}), \gamma(l_j)$ (resp.$\gamma(l_i), \gamma(\overline{l_j})$) of the related vertices are merged if they are different ($\gamma(\overline{l_i}) \neq \gamma(l_j)$ resp. $\gamma(l_i) \neq \gamma(\overline{l_j})$). A merge can be done in constant time by holding a first and last vertex for each component. The affected components are marked "dirty". Dirty components are updated intermittently. Therefore (new) SCCs are removed as described above. Subsequently a new matrix is computed which requires one depth–first search execution on this component. Note that components may also be split when vertices are removed from the graph. This is the case whenever a unit clause has been learnt.

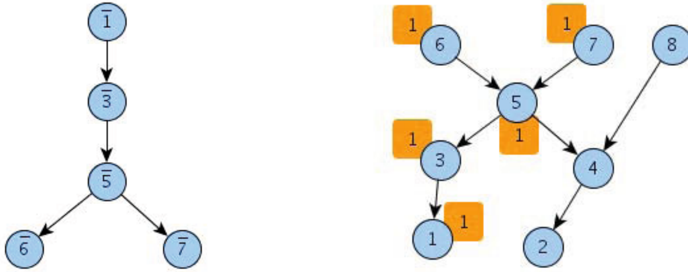
Utilizing matrices in BCP. During usual BCP each clause has two watched literals [30]. Whenever a literal l becomes false all clauses being watched by l are traversed to check whether a clause becomes unit. If for a clause $C \in \mathcal{C}$ another unassigned or true literal $l_n \in C$ can be found, l_n takes over to watch C, otherwise C became unit under the current assignment and Unit Propagation applies as described in 2.1.

At this point we extend the usual Unit Propagation: For a touched clause that is not yet fulfilled by the current partial assignment we check whether the remaining unassigned literals belong to the same component. For early detection of conflicts Unit Propagation is finished first until those clauses are finally checked for extended BCP. If the corresponding reachability matrix indicates that the free literals of one clause have a common successor a breadth–first search is applied to compute the topmost common successors. Note that the matrix may still be misleading since it may happen that all common successors are already assigned.

3.3 A Convenient Alternative

The reachability matrix approach constitutes a feasible way to enhance the broadly used Unit Propagation in SAT solving. However, the maintenance of components and their matrices requires quite some computational effort. Moreover, the compression factor varies for different instances. In this subsection we present an alternative method that approximates the reachability matrix in a practical sense but considerably outperforms the previous approach.

The idea is to cache reachability information on the fly while usual Unit Propagation is performed. Whenever a variable v_i is assigned a value $b \in \{true, false\}$, we first propagate all unit implications that are caused by binary clauses. In this



(a) Unit Propagation of binary clauses (b) sink tags set during Unit Propagation

Fig. 3. The left graph shows the implications of usual Unit Propagation of binary clauses when assigning literal $\overline{1}$. During propagation the *sink-tag* 1 can be set in the implication graph for all literals having opposite polarity to the literals reached by propagation.

Algorithm 2. Assignment of a variable value and caching of sink-tags

```
1 Require Literal l that has to become true, current partial assignment A ;
2 Function assign(l)
3     Q ← {l} /* initialise queue                                          */
4     T ← l̄ /* initialise sink-tag                                          */
5     while Q ≠ ∅ do
6         k ← dequeue(Q) /* remove one element of the queue                */
7         set-sink-tag(k̄, T) /* set sink-tag for opposite literal           */
8         A ← A ∪ k /* assign literal k                                     */
9         foreach j ∈ N(k) : j ∉ Q do enqueue(Q, j) /* enqueue neighbors    */
```

step, values are assigned to exactly those variables v_j whose vertices l_j or $\overline{l_j}$ are successors of l_i if $b = true$ resp. $\overline{l_i}$ if $b = false$ in the implication graph.

Due to the way how edges are created in the implication graph we know that whenever there exists a path from vertex l_i to l_j there is also a path from $\overline{l_j}$ to vertex $\overline{l_i}$ (see section 2). Hence, when Unit Propagation starts from vertex l_i we initialise a *sink-tag* with opposite polarity $\overline{l_i}$. For each vertex l_j that is reached we mark the opposite vertex $\overline{l_j}$ with the current sink-tag. By doing so all vertices that have a path to vertex $\overline{l_i}$ are marked with the corresponding sink-tag. Figure 3 illustrates this approach and Algorithm 2 sketches the procedure. The idea of sink-tags extends the concept of *binary dominators* introduced by Biere for hyper-binary resolution [11,6]. However, binary dominators are attached to variables instead of literals.

Clearly, the sink-tag approach only caches one possible target that can be reached by a vertex. Furthermore, the target does not necessarily have to be a real sink. Thus, this approach does not require the removal or detection of strongly connected components. In Figure 3 the propagation of $\overline{1}$ assigns sink-tag 1 to the vertices shown in the right graph. When asking for a common successor

of vertices 6 and 7 (at a later point in solving) the sink-tag would indicate and even name the correct answer. However, when asking for a common successor of vertices 7 and 8 the sink-tag approach would miss the common successor unless sink-tags are changed by Unit Propagation starting from $\overline{2}$ or $\overline{4}$.

Utilizing sink-tags in BCP Sink-tags can be utilized in a way similar to components and reachability matrices. However, sink-tags represent and replace both - components and bits in a reachability matrix: When a clause is touched during Unit Propagation we additionally check whether all unassigned literals have the same sink-tag. At this point we differentiate two approaches:

optimistic: The state of the sink-tag literal is not considered. If the sink-tag literal is *true* the algorithm still applies breadth–first search to find common successors of the unassigned literals.
pessimistic: The breadth–first search for common successors of the unassigned literals is only applied if the sink-tag literal is unassigned. With this approach one could alternatively just take the sink-tag as one successor and may be ignore any others. However, this dismisses valuable information.

Note that a sink-tag literal l_i of an unassigned variable v can never be $false$ since due to binary implications $\overline{l_i}$ would have implied an assignment to v.

3.4 A Suitable Side Effect

With both approaches it may happen that given a clause $C_1 = \{l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5\}$ and an assignment τ_V with $\tau_V(l_4) = \tau_V(l_5) = false$ the remaining literals have one common successor. This common successor may be equal to one of the literals themselves. E.g. l_1 and l_2 may both imply l_3. Hence, by C_1 and the binary implications (represented in the graph) the clause $C_2 = \{l_3 \vee l_4 \vee l_5\}$ can be resolved. Now C_2 is a subset of C_1 and therefore constitutes a harder constraint than C_1. In general it is said that C_2 *subsumes* C_1. Since C_2 is also valid we can simply remove the literals l_1 and l_2 from C_1 and thus prune the search space. The recognition of such subsumptions comes without any extra computational effort. Figure 4 in the next section indicates that these subsumptions are found quite frequently.

4 Experiments

In this section we present some results of our experiments for the different approaches. We used our solver *SArTagnan*[1] for all tests. It is implemented in C++ and extends the solver *SApperloT* that participated in the SAT competition 2009 [1,24]. Our focus and analysis is not primarily on runtime but mainly on the improvement of Boolean Constraint Propagation in terms of more implications. The tests are performed on 500 *industrial* SAT benchmarks from the

[1] For the SAT Race 2010 SArTagnan was configured to utilise the sink-tag approach in 5 of 8 solving threads. However, for the tests the solver run in sequential mode.

Fig. 4. Comparing different issues of both approaches (upper table). Unit Propagation versus Extended Propagation (lower table).

	Matrix		Opt.Tags		Pess. Tags	
	avg	max	avg	max	avg	max
ext. Prop / Decisions [%]	63.24	1581.93	33.71	1340.64	6.72	226.46
Self Subsumptions [abs]	14038.14	496460	701.54	18340	640.21	28780
Self Subsumptions [%]	9.61	70.97	5.04	96.72	8.14	100
Implied Binaries	16816.36	235042	9100.49	152728	219.17	6238
Implied Units	101.48	2722	146.71	4386	0.53	247

	Unit Prop		Matrix		Opt.Tags		Pess. Tags	
	avg	max	avg	max	avg	max	avg	max
Decisions	976988.0	13099898	550958.7	11454167	835644.2	11636689	925507.49	14192787
Props./Dec.	1145.25	18483.05	1236.46	16809.06	1158.34	16883.87	1175.09	23923.48

SAT competitions and SAT races 2007, 2008 and 2009 [1]. Trivial instances were removed. On average the number of variables per instance is about 115500. Each solver version was given a timeout of 1200 seconds for each instance. Since in both presented approaches the extended propagation is only applied when Unit Propagation has finished we can claim that each variable assigned by extended propagation replaces one decision.

The first line of the upper table in Figure 4 gives a good overview on how many decisions are saved by the different approaches. Apparently the reachability matrix approach outperforms the sink-tag approaches. With the matrix approach the number of variable assignments that are implied by extended propagation are 63 percent of the number of decisions. For both sink-tag approaches the percentage decreases drastically. However, this extended propagation is done without any additional computation.

The second and third line of the upper table of Figure 4 compare the number of subsumptions during extended propagation (cf. Sect. 3.4). The last two lines indicate the number of binary and unit clauses that are created by extended propagation. For the matrix approach there are some more interesting issues. For each instance we measured the biggest matrix that was created for a component. On average this biggest matrix required 870.64 MB without applying Observation 2. The technique behind Observation 2 achieves a large reduction to an average size of 502.47 MB.

Since we are aiming for the reduction of the total number of decisions the lower table of Figure 4 compares the total number of decisions and the implication per decision. Clearly, pure Unit Propagation requires much more decisions than extended propagation using the matrix approach. For the sink-tag approaches the difference decreases but can still be identified.

Even though the matrix approach clearly outperforms the other approaches in terms of quality it has its drawback in the costly maintenance of components and matrices. In Figure 5 extended propagation and Unit Propagation are compared regarding their runtimes. Given the same amount of time the matrix

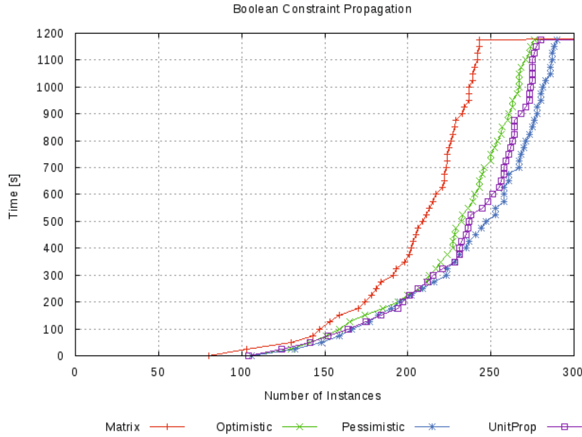


Fig. 5. Each plotted curve represents the performance of one solver on the set of hard benchmarks. One point (x, y) indicates that x instances were solved within y seconds.

approach clearly solves the least number of instances. Due to the cheap computation of sink-tags the alternative approaches perform much better. However the optimistic sink-tag approach cannot compete to highly optimised standard CDCL. For the pessimistic approach it pays off to only apply extended propagation when a valid common successor definitely exists (i.e. unassigned variables that have a common sink-tag l_i that is also unassigned have at least one common and valid implication l_i). It clearly solves more instances than standard CDCL with pure Unit Propagation.

Our results also indicate that there is no significant difference between satisfiable and unsatisfiable instances for the different techniques. The set of benchmarks can be classified into 38 families of instances. For only 4 families ($\approx$ 11 percent) the performance of the matrix technique is significantly worse than CDCL which underlines the potential of our new approaches.

5 Conclusion

Our goal was to reduce the number of branching decisions and to improve the quality of propagation of variable values in SAT solving. In this paper we have presented two approaches on how to extend the broadly used Unit Propagation. The first approach of maintaining reachability matrices shows how the quality of propagation can be improved. The second approach constitutes an inexpensive approximation to the matrix approach. Although not reaching the same quality as the former it clearly outperforms not only the other approaches but also the highly optimised standard CDCL with pure Unit Propagation.

The matrix and the sink-tag approach both consider implications given by binary clauses. However, this can be called static since clauses that become binary

under a partial assignment are not considered. Such clauses would add temporary edges to the implication graph. However, this would require a permanent update of the graph and the matrix. For future research it would be interesting to analyse a more dynamic approach that is not limited to static binary clauses. This will probably further improve extended propagation but it is open how to realise this efficiently.

References

1. The international SAT competition (2002-2009), www.satcompetition.org
2. Aspvall, B., Plass, M.F., Tarjan, R.E.: A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. Inf. Proc. Lett. 8, 121–123 (1979)
3. Audemard, G., Bordeaux, L., Hamadi, Y., Jabbour, S.J., Sais, L.: A generalized framework for conflict analysis. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 21–27. Springer, Heidelberg (2008)
4. Bacchus, F.: Enhancing Davis Putnam with Extended Binary Clause Reasoning. In: 18th AAAI Conference on Artificial Intelligence, pp. 613–619 (2002)
5. Bacchus, F.: Exploring the computational tradeoff of more reasoning and less searching. In: SAT, pp. 7–16 (2002)
6. Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 341–355. Springer, Heidelberg (2004)
7. Beame, P., Kautz, H.A., Sabharwal, A.: Understanding the power of clause learning. IJCAI, 1194–1201 (2003)
8. Berre, D.L.: Exploiting the real power of unit propagation lookahead. Electronic Notes in Discrete Mathematics 9, 59–80 (2001)
9. Biere, A.: Precosat solver description (2009), fmv.jku.at/precosat/preicosat-sc09.pdf
10. Biere, A.: Picosat essentials. JSAT 4, 75–97 (2008)
11. Biere, A.: Lazy hyper binary resolution. In: Algorithms and Applications for Next Generation SAT Solvers, Dagstuhl Seminar 09461, Dagstuhl, Germany (2009)
12. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam (2009)
13. Brafman, R.I.: A simplifier for propositional formulas with many binary clauses. IJCAI, 515–522 (2001)
14. Chu, G., Harwood, A., Stuckey, P.J.: Cache conscious data structures for boolean satisfiability solvers. JSAT 6, 99–120 (2009)
15. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC (1971)
16. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press and McGraw-Hill Book Company (2001)
17. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
18. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)
19. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
20. Han, H., Somenzi, F.: On-the-fly clause improvement. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 209–222. Springer, Heidelberg (2009)
21. Heule, M.: SmArT Solving. PhD thesis, Technische Universiteit Delft (2008)

22. Heule, M., Maaren, H.V.: Aligning cnf- and equivalence-reasoning. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 174–181. Springer, Heidelberg (2005)
23. Kautz, H.A., Selman, B.: Planning as satisfiability. In: Proceedings of the Tenth European Conference on Artificial Intelligence, ECAI 1992, pp. 359–363 (1992)
24. Kottler, S.: Solver descriptions for the SAT competition 2009, satcompetition.org
25. Kottler, S.: SAT Solving with Reference Points. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 143–157. Springer, Heidelberg (2010)
26. Kottler, S., Kaufmann, M., Sinz, C.: Computation of renameable horn backdoors. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 154–160. Springer, Heidelberg (2008)
27. Li, C.M., Anbulagan: Look-ahead versus look-back for satisfiability problems. In: Principles and Practice of Constraint Programming (1997)
28. Marques-Silva, J.P.: Practical Applications of Boolean Satisfiability. In: Workshop on Discrete Event Systems, WODES 2008 (2008)
29. Marques-Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Comput. 48(5), 506–521 (1999)
30. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC (2001)
31. Pipatsrisawat, K., Darwiche, A.: On modern clause-learning satisfiability solvers. J. Autom. Reasoning 44(3), 277–301 (2010)
32. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: Cliques, Coloring, and Satisfiability: DIMACS Implementation Challenge (1993)
33. Van Gelder, A., Tsuji, Y.K.: Satisfiability testing with more reasoning and less guessing. In: Johnson, D.S., Trick, M. (eds.) Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. AMS, Providence (1996)
34. Velev, M.N.: Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In: DATE 2002 (2002)
35. Williams, R., Gomes, C., Selman, B.: Backdoors to typical case complexity. IJCAI (2003)
36. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: ICCAD 2001 (2001)

Designing Difficult Office Space Allocation Problem Instances with Mathematical Programming

Özgür Ülker and Dario Landa-Silva

Automated Scheduling, Optimisation and Planning (ASAP) Research Group
School of Computer Science, University of Nottingham,
Jubilee Campus, Wollaton Road, Nottingham, NG8 1BB, UK
oxu@cs.nott.ac.uk, dario.landasilva@nottingham.ac.uk

Abstract. Office space allocation (OSA) refers to the assignment of room space to a set of entities (people, machines, roles, etc.), with the goal of optimising the space utilisation while satisfying a set of additional constraints. In this paper, a mathematical programming approach is developed to model and generate test instances for this difficult and important combinatorial optimisation problem. Systematic experimentation is then carried out to study the difficulty of the generated test instances when the parameters for adjusting space misuse (overuse and underuse) and constraint violations are subject to variation. The results show that the difficulty of solving OSA problem instances can be greatly affected by the value of these parameters.

Keywords: Office Space Allocation Problem, Integer Programming, Mathematical Modelling, Data Instance Generation, Proof of Optimality.

1 Introduction

We develop a mathematical programming approach to design difficult test instances for the Office Space Allocation (OSA) problem, which is commonly encountered in universities, companies and government institutions. In simple terms, the OSA problem is the task of allocating office spaces (rooms, hallways, etc.) to entities (peoples, machines, roles, etc.) subject to additional constraints. OSA is related to the multiple knapsack [15] and generalised assignment problems [6]. In OSA, the primary goal is to maximise the space utilisation by reducing the misuse of rooms. Misuse of rooms refers to underusing space (under-utilisation of rooms) or overusing space (over-crowding of rooms). Usually, overuse is considered more undesirable than underuse of space. Additional constraints (e.g. people grouping conditions) can arise in different organisations when allocating office space. Any of such constraints can be *hard* (satisfied all the time) or *soft* (desirable but not necessary).

Office space allocation is usually a continuous process due to the constant changes in an organisational environment (departure/arrival of new personnel,

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 280–291, 2011.
© Springer-Verlag Berlin Heidelberg 2011

maintenance/renovation of existing office space, restructuring in organisations etc). This process can involve many conflicting objectives and constraints difficult to tackle when using manual approaches. An automated allocation system can deal with such conflicting objectives and constraints better than a human decision expert especially if the size of the problem grows. An automated system can also provide alternative solutions for different scenarios more quickly. Then, instead of tackling the complex optimisation problem directly, the decision maker can focus on fine tuning the automatically generated allocation based on organisational preferences and requirements.

In this study, the office space allocation problem as described in Landa-Silva [13] is investigated and extended. A 0/1 integer programming model is developed and then Gurobi [10], a commercial integer linear programming (ILP) solver, is applied to solve it. Based on this model, we develop a test instance generator to further investigate the difficulty of the OSA problem through systematic experimentation. The paper is organised as follows: Section 2 outlines previous research on office space allocation. Section 3 presents the mathematical model proposed for OSA while Section 4 describes the test instance generator. Section 5 presents and discusses the results from our experimental study. Finally, Section 6 summarises our contributions and proposes future research directions.

2 Outline of Previous Related Work

One of the earliest works on the optimisation of office space utilisation is that of Ritzman et al. [17], who developed a linear programming model for the distribution of academic offices at the Ohio State University. Benjamin et al. [1] used a linear goal programming model for planning and improving the utilisation of the layout of floor space in a manufacturing laboratory at the University of Missouri Rolla. Giannikos et al. [8] developed a goal programming approach to automate the distribution of offices among staff in an academic institution.

Burke and Varley [5] reported on a questionnaire on the space allocation process in 38 British universities. The emphasis was on the scope of the problem, computing tools to solve it and the constraints in each university. Burke et al. [3] applied hill climbing, simulated annealing [11] and a genetic algorithm [9] to solve the *allocation* (task of creating a complete solution from scratch) and *reorganisation* (task of reallocating entities in a given solution) variants of the problem. The authors used *allocation*, *relocation*, and *swap* operators for moving entities between rooms. Burke et al. [2] later investigated a hybridisation of their previous approaches under a population based framework. The initial solutions were created using a hill climbing operator and then improved using simulated annealing with adaptive cooling schedule. Burke et al. [4] applied multi-objective optimisation [7] to the OSA problem comparing weighted aggregation to Pareto dominance for tackling two objectives: the total space misuse (under/over usage of rooms) and the sum of (soft) constraint violations. They found that these two objectives were conflicting in nature. Later, Landa-Silva and Burke [12] developed an asynchronous cooperative local search method in which local search threads in a population co-operate with each other asynchronously to improve

the overall solution quality. Based on their experiments, the soft constraint *'group by'* was regarded as the most difficult one to satisfy (high number of violations).

Pereira et al. [16] applied a greedy local search and tabu search algorithm to tackle an OSA problem where the goals are to minimise the distance between employees in the same organisation, minimise the office space misallocation and maximise the office space allocation. They reported that tabu search performed better on their OSA problem instances. Lopes and Girimonte [14] analysed a variant of the OSA problem (similar to the one described in [12]) arising in the European Space Agency (ESA). They implemented four types of meta-heuristics: hill climbing, simulated annealing, tabu search and the hybrid meta-heuristic in [2]. To improve the performance of these algorithms, variations to the local search, and new constraints management algorithms were designed by the authors.

3 Mathematical Programming Model

An earlier version of the following model was presented in [18]. The set of rooms is denoted by R and the set of entities is denoted by E. The size of entity e is S_e and the capacity of room r is C_r. There is a matrix X of $|R| \times |E|$ binary decision variables where each $x_{er} = 1$ if entity e is allocated to room r, otherwise $x_{er} = 0$. Let A be the adjacency list of $|R|$ adjacency vectors each one denoted by A_r and holding the list of rooms adjacent to room r. Similarly, let N be the nearby list of $|R|$ nearby vectors each one denoted by N_r and holding the list of rooms near to room r. The adjacency vector A_r for a room r is usually quite smaller compared to the nearby vector N_r, i.e. more rooms are considered to be 'near' to room r than considered to be 'adjacent' to the same room.

There are ten requirements or constraints handled here. Most of these constraints can be set as *hard* (must be satisfied) or *soft* (desirable to satisfy) in our formulation. In other words, when a constraint is set as soft, minimising its violation becomes an objective in the problem formulation. The exception here is the *'All allocated'* constraint (all entities must be allocated) which is always enforced. The next subsections present these alternative formulations in the constraint set and in the objective function. For each constraint type (defined below), HC^{al}, HC^{na}, HC^{sr}, HC^{nsr}, HC^{nsh}, HC^{ad}, HC^{gr}, HC^{aw}, HC^{cp} denote the corresponding constraint as *hard* while SC^{al}, SC^{na}, SC^{sr}, SC^{nsr}, SC^{nsh}, SC^{ad}, SC^{gr}, SC^{aw}, SC^{cp} denote the corresponding constraint as *soft*. Note that each *soft* constraint is associated with a binary indicator variable y^{cst} which is set to 1 if the respective *soft* constraint is violated. Some constraint types require additional binary variables (y_r^{cst}) over $r \in R$.

3.1 Modeling Hard Constraints

All allocated: each entity $e \in E$ must be allocated to exactly one room $r \in R$.

$$\sum_{r \in R} x_{er} = 1 \quad \forall e \in E \tag{1}$$

Allocation: entity e to be placed into room r. ($(e, r) \in HC^{\text{al}}$).

$$x_{er} = 1 \tag{2}$$

Non allocation: entity e not to be placed into room r. ($(e, r) \in HC^{\text{na}}$).

$$x_{er} = 0 \tag{3}$$

Same room: entities e_1 and e_2 to be placed into same room. ($(e_1, e_2) \in HC^{\text{sr}}$).

$$x_{e_1 r} = 1 \leftrightarrow x_{e_2 r} = 1 \quad \forall r \in R \quad \text{i.e.}$$

$$x_{e_1 r} - x_{e_2 r} = 0 \quad \forall r \in R \tag{4}$$

Not in same room: entities e_1 and e_2 to be placed into different rooms. ($(e_1, e_2) \in HC^{\text{nsr}}$).

$$x_{e_1 r} = 1 \leftarrow x_{e_2 r} = 0 \quad \forall r \in R \quad \text{i.e.}$$

$$x_{e_1 r} + x_{e_2 r} \leq 1 \quad \forall r \in R \tag{5}$$

Not sharing: entity e not to share a room with any other entity. ($(e) \in HC^{\text{nsh}}$).

$$x_{er} = 1 \rightarrow \sum_{f \in E - e} x_{fr} = 0 \quad \forall r \in R \quad \text{i.e.}$$

$$\sum_{f \in E - e} x_{fr} \leq (|E| - 1) - (|E| - 1) x_{er} \quad \forall r \in R \tag{6}$$

Adjacency: entities e_1 and e_2 placed into adjacent rooms. ($(e_1, e_2) \in HC^{\text{ad}}$).

$$x_{e_1 r} = 1 \rightarrow \sum_{s \in A_r} x_{e_2 s} = 1 \quad \forall r \in R \quad \text{i.e.}$$

$$x_{e_1 r} \leq \sum_{s \in A_r} x_{e_2 s} \leq 1 \quad \forall r \in R \tag{7}$$

Group by: entities in a group placed near to the group head f. ($(e, f) \in HC^{\text{gr}}$).

$$x_{er} = 1 \rightarrow \sum_{s \in N_r} x_{fs} = 1 \quad \forall r \in R \quad \text{i.e.}$$

$$x_{er} \leq \sum_{s \in N_r} x_{fs} \leq 1 \quad \forall r \in R \tag{8}$$

Away from: entities e_1 and e_2 to be placed in rooms away from each other. ($(e_1, e_2) \in HC^{\text{aw}}$).

$$x_{e_1 r} = 1 \rightarrow \sum_{s \in N_r} x_{e_2 s} = 0 \quad \forall r \in R \quad \text{i.e}$$

$$0 \leq \sum_{s \in N_r} x_{e_2 s} \leq 1 - x_{e_1 r} \quad \forall r \in R \tag{9}$$

Capacity: Room r must not be overused. ($(r) \in HC^{\text{cp}}$).

$$\sum_{e \in E} S_e x_{er} \leq C_r \tag{10}$$

3.2 Modeling Constraints as Objectives

Allocation: indicator variable $y^{\text{al}}(i)$ is set if $SC^{\text{al}}(i)$ is not satisfied.

$$y^{\text{al}}(i) = 1 - x_{er} \tag{11}$$

Non allocation: indicator variable $y^{\text{na}}(i)$ is set if $SC^{\text{na}}(i)$ is not satisfied.

$$y^{\text{na}}(i) = x_{er} \tag{12}$$

Same room: indicator variable $y^{\text{sr}}(i)$ is set if $SC^{\text{sr}}(i)$ is not satisfied.

$$2y_r^{\text{sr}}(i) - 1 \leq x_{e_1 r} - x_{e_2 r} \leq 1 - \epsilon + \epsilon y_r^{\text{sr}}(i) \quad \forall r \in R \tag{13}$$

$$y^{\text{sr}}(i) = \sum_{r \in R} y_r^{\text{sr}}(i) \tag{14}$$

Not in same room: indicator variable $y^{\text{nsr}}(i)$ is set if $SC^{\text{nsr}}(i)$ is not satisfied.

$$(1 + \epsilon) - (1 + \epsilon) y_r^{\text{nsr}}(i) \leq x_{e_1 r} + x_{e_2 r} \leq 2 - y_r^{\text{nsr}}(i) \quad \forall r \in R \tag{15}$$

$$y^{\text{nsr}}(i) = \sum_{r \in R} (1 - y_r^{\text{nsr}}(i)) \tag{16}$$

Not sharing: indicator variable $y^{\text{nsh}}(i)$ is set if $SC^{\text{nsh}}(i)$ is not satisfied.

$$(|E| - 1)(2 - x_{er} - y_r^{\text{nsh}}(i)) \geq \sum_{f \in E - e} x_{fr} \quad \forall r \in R \tag{17}$$

$$\sum_{f \in E - e} x_{fr} \geq (|E| - 1)(1 - x_{er}) + \epsilon - (|E| - 1 + \epsilon) y_r^{\text{nsh}}(i) \quad \forall r \in R \tag{18}$$

$$y^{\text{nsh}}(i) = \sum_{r \in R} (1 - y_r^{\text{nsh}}(i)) \tag{19}$$

Adjacency: indicator variable $y^{\text{ad}}(i)$ is set if $SC^{\text{ad}}(i)$ is not satisfied.

$$y_r^{\text{ad}}(i) + x_{e_1 r} - 1 \leq \sum_{s \in A_r} x_{e_2 s} \leq x_{e_1 r} - \epsilon + (1 + \epsilon) y_r^{\text{ad}}(i) \quad \forall r \in R \tag{20}$$

$$y^{\text{ad}}(i) = \sum_{r \in R}(1 - y_r^{\text{ad}}(i)) \tag{21}$$

Group by: indicator variable $y^{\text{gr}}(i)$ is set if $SC^{\text{gr}}(i)$ is not satisfied.

$$y_r^{\text{gr}}(i) + x_{er} - 1 \leq \sum_{s \in N_r} x_{fs} \leq x_{er} - \epsilon + (1+\epsilon)y_r^{\text{gr}}(i) \quad \forall r \in R \tag{22}$$

$$y^{\text{gr}}(i) = \sum_{r \in R}(1 - y_r^{\text{gr}}(i)) \tag{23}$$

Away from: indicator variable $y^{\text{aw}}(i)$ is set if $SC^{\text{aw}}(i)$ is not satisfied.

$$1 - x_{e_1 r} + \epsilon - (1+\epsilon)y_r^{\text{aw}}(i) \leq \sum_{s \in N_r} x_{e_2 s} \leq 2 - x_{e_1 r} - y_r^{\text{aw}}(i) \quad \forall r \in R \tag{24}$$

$$y^{\text{aw}}(i) = \sum_{r \in R}(1 - y_r^{\text{aw}}(i)) \tag{25}$$

Capacity: indicator variable $y^{\text{cp}}(i)$ is set if $SC^{\text{cp}}(i)$ is not satisfied.

$$\sum_{e \in E} S_e x_{er} + (C_r + \epsilon)(1 - y^{\text{cp}}(i) \geq C_r + \epsilon \tag{26}$$

$$\sum_{e \in E} S_e x_{er} + (\sum_{e \in E} S_e - C_r)(1 - y^{\text{cp}}(i) \leq \sum_{e \in E} S_e \tag{27}$$

3.3 Objective Function

The objective function is the weighted sum of the space misuse ($underuse + 2 \cdot overuse$) and the soft constraints violation penalty. The penalties associated to each soft constraint type are: w^{al}, w^{na}, w^{sr}, w^{nsr}, w^{nsh}, w^{ad}, w^{gr}, w^{aw}, and w^{cp}. The objective function Z to minimise is given by:

$$\begin{aligned} Z = \sum_{r \in R} max\left(C_r - \sum_{e \in E} x_{er}S_e \,,\; 2\sum_{e \in E} x_{er}S_e - C_r\right) + w^{\text{al}} \sum_{i=1}^{|SC^{\text{al}}|} y^{\text{al}}(i) \\ + w^{\text{na}} \sum_{i=1}^{|SC^{\text{na}}|} y^{\text{na}}(i) + w^{\text{sr}} \sum_{i=1}^{|SC^{\text{sr}}|} y^{\text{sr}}(i) + w^{\text{nsr}} \sum_{i=1}^{|SC^{\text{nsr}}|} y^{\text{nsr}}(i) + w^{\text{nsh}} \sum_{i=1}^{|SC^{\text{nsh}}|} y^{\text{nsh}}(i) \\ + w^{\text{ad}} \sum_{i=1}^{|SC^{\text{ad}}|} y^{\text{ad}}(i) + w^{\text{gr}} \sum_{i=1}^{|SC^{\text{gr}}|} y^{\text{gr}}(i) + w^{\text{aw}} \sum_{i=1}^{|SC^{\text{aw}}|} y^{\text{aw}}(i) + w^{\text{cp}} \sum_{i=1}^{|SC^{\text{cp}}|} y^{\text{cp}}(i) \end{aligned} \tag{28}$$

4 Test Instance Generator for OSA

We have access to some real-world data for the OSA problem but in order to systematically investigate this problem, we developed a test instance generator based on the mathematical programming approach. The generator currently supports the nine types of constraints described in the previous section, and the generation of entities, rooms, and floor layout. An outline of the generator is shown in Algorithm 1. The generator algorithm starts with the creation of entities, groups (set of entities) and initial sets of *hard* and *soft* constraints. Then it creates or modifies the floor layout and/or the room sizes by means of a constructive heuristic. The generator tries to 'plant' a core solution into the instance by allocating entities into rooms according to the initial constraint sets. In order to experimentally investigate the difficulty of the created test instances, four parameters directly related to space misuse (overuse/underuse) and to soft constraint violations were devised. These parameters are:

Algorithm 1. OSA Test Instance Generator Algorithm

Input: input file of parameters.
Output: data instance.
— Create Groups, Entities, Floor Layout and Constraints.
— Placement of entities according to the *hard* constraints.
— Calculate space that is minimally required for each room.
— Placement of entities according to the *soft* constraints.
— Room size adjustments via (positive or negative) slack amounts.
— Post processing

1. *Slack Space Rate:* After all the entities are allocated to rooms, this parameter determines whether the room size will be modified. This parameter adjusts the amount of space misuse.
2. *Negative Slack Amount:* Is the amount by which room capacity is reduced and is determined by a percentage of the total sizes of the entities already allocated to rooms. This parameter adjusts the amount of space overuse.
3. *Positive Slack Amount:* Is the amount by which room capacity is increased and is determined by a percentage of the total sizes of the entities already allocated to rooms. This parameter adjusts the amount of space underuse.
4. *Violation Rate:* When allocating entities to rooms as indicated by the soft constraints, there might be some violations of constraints, i.e. conflicts. A soft constraint is removed from the constraint set with this rate if such a conflict occurs. This parameter adjusts the violation of soft constraints.

5 Experiments and Results

Six real-world benchmark instances (called Nott) taken from [13] were used for experimentation here. Additional experiments were carried using test instances created with our generator as well.

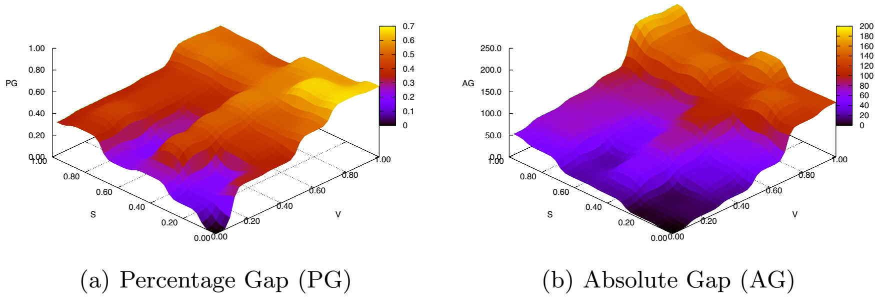
Table 1. Constraint penalties for the best results obtained for each problem instance of the Nott Dataset

	Nott1	**Nott1***	**Nott1b**	**Nott1c**	**Nott1d**	**Nott1e**	**Wolver**
Allocation	40.00	20.00	0.00	40.00	0.00	0.00	0.00
Same Room	0.00	0.00	80.00	0.00	0.00	0.00	0.00
Not Sharing	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Adjacency	10.00	10.00	0.00	10.00	0.00	0.00	0.00
Group by	11.18	22.36	11.18	11.18	11.18	0.00	0.00
Away From	20.00	30.00	0.00	20.00	0.00	40.00	0.00
Constraint Penalty	81.18	82.36	91.18	81.18	11.18	40.00	0.00
Overuse	130.80	106.40	64.20	182.90	164.70	13.80	486.04
Underuse	134.20	122.00	87.90	41.65	26.85	123.90	148.15
Usage Penalty	265.00	228.40	152.10	224.55	191.55	137.70	634.19
Total Penalty	346.18	310.76	243.28	305.73	202.73	177.70	634.19
Lower Bound	201.86	273.16	131.45	305.73	202.73	177.70	634.19
Percentage Gap	%41.70	%12.10	%46.00	%0.00	%0.00	%0.00	%0.00

To solve the 0/1 IP formulation, Gurobi 3.0.1 [10] was used on a PC with a Core 2 Duo E8400 3Ghz processor and 2GB of RAM. Each problem instance was given 30 minutes of solver runtime. The following penalties were used for each soft constraint violation: $w^{\mathrm{al}} = 20$, $w^{\mathrm{na}} = 10$, $w^{\mathrm{sr}} = 10$, $w^{\mathrm{nsr}} = 10$, $w^{\mathrm{nsh}} = 50$, $w^{\mathrm{ad}=10}$, $w^{\mathrm{gr}} = 11.18$ for the Nott instances, $w^{\mathrm{gr}} = 10$ for the generated test instances, $w^{\mathrm{aw}} = 10$, and $w^{\mathrm{cp}} = 10$.

Table 1 summarises the best results obtained after a run of 30 minutes on each problem instance (from Nott1 to Wolver). Note that these dataset instances do not contain *non-allocation*, *not in same room*, or *capacity* constraints, the other six constraint types are present in these real-world instances. The penalties for each constraint violation are given in rows 2-7 of the table. Two different experiments were run on the largest problem instance Nott1. It was observed during experiments that minimising *same room* constraint violations is the most difficult, especially for the Nott1b instance (value of 80.00). So, we conducted an additional experiment for tackling the *same room* constraint in the Nott1 instance (largest one). In Nott1 column, *same room* constraints were all set as soft, whereas in column Nott1*, *same room* constraints were all set as *hard*. Notice that this latter setup achieved a lower usage penalty by roughly 35 square meters. We can see that in all these instances, the constraint penalty turned out to be significantly lower than the usage penalty. For all these real-world instances, our model and solution approach produced the best results in the literature so far [18,12]. Table 1 also shows that for instances Nott1c, Nott1d, Nott1e and Wolver we obtained optimal results while instances Nott1 and Nott1b remain very challenging.

Our next experiments focused on studying the difficulty of the generated test instances by changing the four generator parameters described in Section 4: *slack space rate* (S), *positive slack amount* (P), *negative slack amount* (N) and *violation rate* (V). The term ‘difficulty’ in this paper refers to the difficulty of



(a) Percentage Gap (PG)

(b) Absolute Gap (AG)

Fig. 1. The effect of changing S and V on the percentage and absolute gaps

the optimality proof for the ILP solver (i.e. the gap or difference between the best found solution and the best found lower-bound on the optimal solution). The aim of this experimentation was to check if incrementing the above four parameters had any effect on this gap and hence the optimality proof difficulty of the test instances. In generating all our test instances, the generator used the same entity set (with 150 entities), same initial hard constraint set, and same initial soft constraint set (subject to modifications of the V parameter). The S, P, and N parameters were varied to adjust the size of the rooms (92 rooms in each instance) to obtain various amounts of space misuse (underuse/overuse). We created two different datasets. In the $SVe150$ dataset, the *slack space rate* (S) and *violation rate* (V) were varied between 0.0 to 1.0 with 0.2 increments (i.e. 36 test instances). In the $PNe150$ dataset, the *positive slack amount* (P) and *negative slack amount* (N) were varied between 0.00 to 0.25 with 0.05 increments (i.e. 36 more instances).

Table 2(a) shows results for the $SVe150$ dataset. Columns S and V represent the amount of *slack space* and *violation* rates respectively. Columns C, B and % represent the objective value achieved, the lower bound on the objective value and the percentage gap between the objective value and the lower bound respectively. The *positive slack* (P) and *negative slack* (N) amounts were fixed at 0.10 for this experiment. It was observed that although increasing S and V individually increases the percentage gaps, this increase tends to stabilize (and in fact decreases) after certain levels of S and V. It was observed that the percentage gaps tend to peak around $S = 0.4$ and $V = 0.8$. The absolute gaps (the raw difference between the bound and obtained objective value) exhibit a somehow expected smooth increase with larger S and V values. Table 2(b) shows results for the $PNe150$ dataset. The effect of changing the *positive slack* (P) and *negative slack* (N) amounts on the achieved objective values, lower bounds and percentage gaps obtained is observed in this table. The *slack rate* (S) and *violation rate* (V) were fixed at 0.5. Figures 1 and 2 illustrate graphically the effect of S, V, P and N in our experimental results.

The percentage gaps obtained in our experiments serve as an evidence that our generator is able to create difficult test instances. i.e. with significant high percentage gaps between the achieved objective values and the lower bounds

Table 2. The objective values, the lower bounds and the percentage gaps obtained when solving the $SVe150$ and $PNe150$ generated instances under different values for parameters S, V, P and N

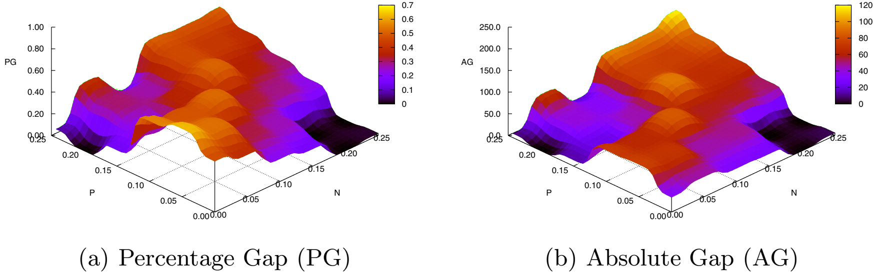
(a) Changing Slack Space Rate (S) and Violation Rate (V)

S	V	C	B	%
0.00	0.00	0.00	0.00	0.0%
0.00	0.20	32.00	21.10	34.0%
0.00	0.40	57.00	37.30	34.5%
0.00	0.60	95.50	46.20	51.6%
0.00	0.80	171.00	55.90	67.3%
0.00	1.00	193.00	67.40	65.1%
0.20	0.00	28.10	24.40	13.2%
0.20	0.20	52.90	43.20	18.3%
0.20	0.40	86.60	51.40	40.6%
0.20	0.60	122.80	62.20	49.3%
0.20	0.80	212.70	72.10	66.1%
0.20	1.00	210.30	86.20	59.0%
0.40	0.00	82.80	62.00	25.1%
0.40	0.20	116.30	63.70	45.2%
0.40	0.40	155.10	77.20	50.3%
0.40	0.60	188.80	84.70	55.1%
0.40	0.80	208.70	94.10	54.9%
0.40	1.00	271.50	107.80	60.3%
0.60	0.00	109.70	87.10	20.6%
0.60	0.20	129.70	107.00	17.5%
0.60	0.40	168.20	121.90	27.5%
0.60	0.60	205.20	129.50	36.9%
0.60	0.80	289.10	138.80	52.0%
0.60	1.00	278.70	147.60	47.1%
0.80	0.00	124.70	76.80	38.4%
0.80	0.20	160.30	89.20	44.4%
0.80	0.40	173.60	101.90	41.3%
0.80	0.60	195.90	114.70	41.5%
0.80	0.80	267.80	122.20	54.4%
0.80	1.00	276.10	134.80	51.2%
1.00	0.00	169.10	111.00	34.4%
1.00	0.20	194.20	124.10	36.1%
1.00	0.40	221.40	141.40	36.1%
1.00	0.60	243.40	149.90	38.4%
1.00	0.80	340.40	157.50	53.7%
1.00	1.00	345.30	165.90	51.9%

(b) Changing Positive Slack (P) and Negative Slack (N) Amounts

P	N	C	B	%
0.00	0.00	73.00	38.59	47.13%
0.00	0.05	119.40	72.70	39.11%
0.00	0.10	145.20	114.29	21.28%
0.00	0.15	186.00	156.88	15.65%
0.00	0.20	210.20	209.90	0.14%
0.00	0.25	250.80	244.74	2.42%
0.05	0.00	119.90	44.90	62.55%
0.05	0.05	130.80	61.06	53.32%
0.05	0.10	141.40	95.47	32.48%
0.05	0.15	185.00	136.33	26.31%
0.05	0.20	202.40	202.40	0.00%
0.05	0.25	232.70	232.70	0.00%
0.10	0.00	130.40	57.67	55.78%
0.10	0.05	134.00	70.09	47.70%
0.10	0.10	162.60	78.64	51.64%
0.10	0.15	176.30	118.26	32.92%
0.10	0.20	205.00	146.51	28.53%
0.10	0.25	240.60	188.26	21.76%
0.15	0.00	96.00	81.50	15.10%
0.15	0.05	105.40	83.25	21.01%
0.15	0.10	124.50	79.35	36.27%
0.15	0.15	183.50	90.22	50.83%
0.15	0.20	192.90	126.09	34.63%
0.15	0.25	229.50	160.13	30.23%
0.20	0.00	108.60	108.60	0.00%
0.20	0.05	135.40	95.37	29.56%
0.20	0.10	129.90	95.47	26.51%
0.20	0.15	167.20	90.65	45.78%
0.20	0.20	177.20	102.75	42.02%
0.20	0.25	219.50	134.46	38.74%
0.25	0.00	126.00	116.21	7.77%
0.25	0.05	180.30	110.82	38.54%
0.25	0.10	132.90	113.27	14.77%
0.25	0.15	192.40	96.33	49.93%
0.25	0.20	195.30	100.71	48.43%
0.25	0.25	233.00	115.48	50.44%

provided by the solver. One interesting observation is that the difficulty of the test instances is not necessarily affected by increasing or decreasing P and N independently. The percentage gaps were usually highest when P and N were set equal or close to each other. Also, the percentage gaps were usually lower



(a) Percentage Gap (PG) (b) Absolute Gap (AG)

Fig. 2. The effect of changing P and N on the percentage and absolute gaps

when the absolute value gap between P and N was increased and the gaps were minimal when $N - P$ was the highest. The absolute value gap between the objective value and the bounds exhibited a similar pattern to the percentage gap case. This can be attributed to the fact that when N is increased, the resulting test instance has a lot of rooms with less available space than required (high overuse) but not enough rooms with extra capacity to compensate the lack of space if P is kept low. At this setting, the solver is expected to immediately allocate any extra space it can find and the remaining time when solving the instance is concentrated on minimizing the overuse. However, when P and N are kept close to each other, there are enough rooms with both overuse and underuse to compensate for each other, hence the solver has to make choices to minimize overuse, underuse and constraint violations, spending more computation time as a result.

6 Conclusions

In this work, a 0/1 IP formulation was proposed to model various *hard* and *soft* constraints in the office space allocation (OSA) problem. This model was implemented in the Gurobi solver and we improved the best results obtained so far for the existing Nott dataset. A test instance generator was also described here and further experiments were carried out on new test instances generated. Our experiments focused on studying the effect that four different parameters of the generator, which affect the space misuse (underuse/overuse) and the soft constraint violations, have on the percentage and absolute gaps between the achieved objective value and the lower-bound on the optimal solution found by the solver. It was observed that an important factor affecting the optimality proof difficulty of the test instances, was the difference between *negative slack* (N) and *positive slack* (P) amounts, which adjust the *overuse* and *underuse* of rooms respectively in the generated test instances. Although raising the *slack space rate* (S) and *violation rate* (V) increased the percentage gaps, their effect was less prominent than the effect of N and P. Future research will concentrate on a detailed study of the effect of these four parameters on the overuse, underuse and constraint violation penalties. We also intend to develop more effective solution techniques to tackle the most difficult instances like Nott1b and Nott1.

References

1. Benjamin, C., Ehie, I., Omurtag, Y.: Planning facilities at the university of missouri-rolla. Interfaces 22(4) (1992)
2. Burke, E.K., Cowling, P., Landa Silva, J.D.: Hybrid population-based metaheuristic approaches for the space allocation problem. In: Proceedings of the 2001 Congress on Evolutionary Computation (CEC 2001), pp. 232–239 (2001)
3. Burke, E.K., Cowling, P., Landa Silva, J.D., McCollum, B.: Three methods to automate the space allocation process in UK universities. In: Burke, E., Erben, W. (eds.) PATAT 2000. LNCS, vol. 2079, pp. 254–273. Springer, Heidelberg (2001)
4. Burke, E.K., Cowling, P., Landa Silva, J.D., Petrovic, S.: Combining hybrid metaheuristics and populations for the multiobjective optimisation of space allocation problems. In: Proceedings of the 2001 Genetic and Evolutionary Computation Conference (GECCO 2001), pp. 1252–1259 (2001)
5. Burke, E.K., Varley, D.B.: Space allocation: An analysis of higher education requirements. In: Burke, E.K., Carter, M. (eds.) PATAT 1997. LNCS, vol. 1408, pp. 20–33. Springer, Heidelberg (1998)
6. Cattrysse, D.G., Van Wassenhove, L.N.: A survey of algorithms for the generalized assignment problem. European Journal of Operational Research 60(3), 260–272 (1992)
7. Coello Coello, C.A., Lamont, G.B., Van Veldhuizen, D.A.: Evolutionary Algorithms for Solving Multi-Objective Problems, 2nd edn. Springer, Heidelberg (2006)
8. Giannikos, J., El-Darzi, E., Lees, P.: An integer goal programming model to allocate offices to staff in an academic instituition. Journal of the Operational Research Society 46(6), 713–720 (1995)
9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning, 1st edn. Addison-Wesley, Reading (1989)
10. Gurobi Optimization: Gurobi (2010), http://www.gurobi.com
11. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220, 671–680 (1983)
12. Landa-Silva, D., Burke, E.K.: Asynchronous cooperative local search for the office-space-allocation problem. INFORMS J. on Computing 19(4), 575–587 (2007)
13. Landa-Silva, J.D.: Metaheuristics and Multiobjective Approaches for Space Allocation. Ph.D. thesis, School of Computer Science and Information Technology, University of Nottingham (2003)
14. Lopes, R., Girimonte, D.: The office-space-allocation problem in strongly hierarchized organizations. In: Cowling, P., Merz, P. (eds.) EvoCOP 2010. LNCS, vol. 6022, pp. 143–153. Springer, Heidelberg (2010)
15. Martello, S., Toth, P.: Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc., New York (1990)
16. Pereira, R., Cummiskey, K., Kincaid, R.: Office space allocation optimization. In: IEEE Systems and Information Engineering Design Symposium (SIEDS 2010), pp. 112–117 (2010)
17. Ritzman, L., Bradford, J., Jacobs, R.: A multiple objective approach to space planning for academic facilities. Managament Science 25(9), 895–906 (1980)
18. Ülker, O., Landa-Silva, D.: A 0/1 integer programming model for the office space allocation problem. Electronic Notes in Discrete Mathematics 36, 575–582 (2010)

Speed Dating
An Algorithmic Case Study Involving Matching and Scheduling

Bastian Katz, Ignaz Rutter, Ben Strasser, and Dorothea Wagner

Faculty of Informatics, Karlsruhe Institute of Technology (KIT), Germany
{bastian.katz,ignaz.rutter,dorothea.wagner}@kit.edu,
mail@ben-strasser.net

Abstract. In this paper we formalize and solve the *speed dating problem*. This problem arises in the context of speed dating events, where several people come together to date each other for a short time. For larger events of this type it is not possible to have each possible pair of persons meet. Instead, based on forms filled out by the participants, the organizer of such an event decides in advance, which pairs of people should meet and also schedules the times of their dates. Moreover, since people pay for participating in such events, aside from the overall quality of the dates, it is important to find a *fair* schedule, where people from the same group (e.g., all women) have a comparable number of dates.

We model the organizer's problem for speed dating, study its complexity and design efficient algorithms for solving it. Finally, we present an experimental evaluation and show that our algorithms are indeed able to solve realistic problem instances within a reasonable time frame.

1 Introduction

A speed dating event is an event, where people that wish to find a partner, come and date each other for a short period of time. When such events came up they were usually small enough that during one evening every potential pair could have a date. Due to the increasing size of such events this is no longer feasible and the organizer has to schedule in advance who meets whom, and when. The schedule is usually set up in rounds such that in each round each person dates at most one other person. Sometimes it is not possible that everybody has a date in every round, for example the ratio of men to women usually is around 3:2, and hence some men need to skip a round now and then.

To come up with potential matches, the organizer asks the participants to fill out forms about their ideal partner before the event. Based on this information he estimates a certain quality for each date. Certain dates have such a poor quality that they would only upset the participants and therefore should in no case take place. One approach to solving the speed dating problem would now be to select a set of dates, such that the total quality of all dates is maximized and no person has more dates than the number of rounds. This allows to limit the number of rounds and thus guarantees a fixed time frame for the whole event.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 292–303, 2011.
© Springer-Verlag Berlin Heidelberg 2011

There are a few problems to consider when choosing the set of pairs that meet each other and an according schedule. First of all, in each round every person can be involved in at most one date, i.e., for each round, the set of dates forms a matching of the people. Some persons may be more attractive than others. When maximizing the total quality of all dates, they would probably get a lot of dates, possibly at the cost of other people. Since all people participating in a speed dating event pay for the registration, it is crucial to find a fair distribution of dates. However, a completely fair distribution is not always feasible. For example there generally are more men than women participating in such events. The only way to deal with this problem is to make some men skip some of the rounds. Again these should be fairly distributed among all men. If there are n men and m women with $n > m$, σ rounds yield a total of $m\sigma$ dates. Hence every man should get roughly $m/n \cdot \sigma$ dates. As $\sigma \cdot m$ may not be divisible by n it would be reasonable to require for every man at least $\lfloor m/n \cdot \sigma \rfloor$ and at most $\lceil m/n \cdot \sigma \rceil$ dates. We will however introduce more flexible bounds that allow an upper and lower bound for each vertex individually. In this way more elaborate constraints can be modeled, such as VIP persons who pay more and are less likely to skip rounds. Finally, it is important that a solution can be computed quickly as the organizer of such events may wish to perform the computation of the schedule as close to the start of the actual event as possible, in order to accomodate for late registrations or people that register but do not show up.

So far we have only described *bipartite speed dating*, where the people are divided into two groups (men and women) and all dates take place between people from different groups. We also consider the *general speed dating* problem, where we do not make this distinction between men and women and allow arbitrary dates to take place. This more general setting could also be applied in other settings, for example at scientific meetings in Dagstuhl people are usually randomly assigned to tables for dinner. To facilitate an efficient communication among researchers, it may be desirable to determine this assignment based on a network representing common research topics, instead. Note that the bipartite speed dating problem is a special case of the general speed dating problem, where the dates between persons of the same gender are rated very badly. We will see later that these two problems behave quite differently in terms of computational complexity.

Outline and Contribution. We introduce the problem of scheduling meetings of pairs in a group of people, as they arise for example in speed dating events. Based on the already identified criteria total quality and fairness, we derive a precise problem formulation for the speed dating problem. We show that the general problem is NP-hard and give a polynomial-time algorithm for the bipartite case. We further show that the general case admits a polynomial-time algorithm that simultaneously approximates the total quality and the fairness violation. Finally, we demonstrate the effectiveness and efficiency of our algorithms with an experimental evaluation that compares the performances of our algorithms with a greedy solution on a variety of problem instances, among them randomly generated instances and real-world instances of social networks.

The paper is organized as follows. First, we formalize the speed dating problem and study its complexity in Section 2. In Section 3 we give a polynomial-time algorithm for the bipartite case and modify it into an approximation algorithm for the general case in Section 4. We present our experimental evaluation and our conclusions in Section 5.

2 Preliminaries

In this section we formalize the speed dating problem, arrive at a formal problem statement, and consider its complexity status. Along the way we introduce notations that we use throughout this paper.

The input to the general speed dating problem consists of a tuple (G,q,ℓ,h,σ), where $G=(V,E)$ is an undirected graph with weights $q\colon E \longrightarrow \mathbb{N}\setminus\{0\}$ and two functions $\ell,h\colon V \longrightarrow \mathbb{N}_0$ specifying lower and upper bounds for the number of dates for each vertex such that $\ell(v) \le h(v) \le \deg(v)$ holds for all $v \in V$, as well as a number of rounds σ.

A *feasible solution* to the speed dating problem can be encoded as a subgraph $G'=(V,E')$ of G with the property that $\ell(v) \le \deg_{G'}(v) \le h(v)$ for all $v \in V$ along with a *proper coloring* of the edges that uses at most σ colors. A proper coloring of the edges is such that any two edges sharing a vertex have distinct colors. The *quality* of a feasible solution can be measured by the total weight of its edges, i.e., $q(G')=q(E')=\sum_{e\in E'} q(e)$. For a solution G' of the speed dating problem we also write $\mathrm{dates}(v)$ instead of $\deg_{G'}(v)$ to denote the number of dates a vertex v participates in.

The main problem with this approach is that, depending on the structure of the input graph G, a feasible solution may not even exist as it may not be possible to find a solution with $\mathrm{dates}(v) \ge \ell(v)$ for all $v \in V$. We therefore relax this lower bound on the number of dates and consider any solution G' that is properly σ-edge colored and satisfies $\mathrm{dates}(v) \le h(v)$ for all $v \in V$. We measure the quality of such a solution in terms of the quality as introduced before and by its *fairness violation* $\delta(G')$. The fairness violation $\delta(v)$ of a single vertex v is the amount to which its lower bound is violated or 0 if it is satisfied, i.e., $\delta(v)=\max\{\ell(v)-\mathrm{dates}(v),0\}$. The overall fairness violation of a solution is the maximum fairness violation among all vertices, i.e., $\delta(G')=\max_{v\in V}\delta(v)$.

In a nutshell, the fairness violation δ describes the degree of fairness of a given solution and the quality q describes the overall quality of the selected dates. Since customer satisfaction is a priority, we focus on minimizing the fairness violation first and on optimizing the quality of the dates as a second priority. We are now ready to formally state the speed dating problem.

Given an instance (G,q,ℓ,h,σ), the problem SPEEDDATING asks to find a solution G' that minimizes the fairness violation $\delta(G')$ and among all such solutions has the maximum quality $q(G')$. The bipartite speed dating problem is defined analogously, except that the graph G is bipartite. In the following we will assume that $\max_{v\in V} h(v) \le \sigma$ as it never makes sense to allow more dates than rounds for any person. Unfortunately, the general problem is NP-complete.

Theorem 1. SPEEDDATING *is NP-complete, even if* $\sigma = 3$.

The proof is by reduction from EDGECOLORING, we omit it due to space constraints. Although this problem is NP-hard in its general form, solving the speed dating problem is not completely hopeless since EDGECOLORING admits solutions for quite some interesting cases. First of all, for bipartite graphs Δ colors always suffice and a solution can be computed efficiently [2]. Second, every graph can be colored with at most $\Delta+1$ colors [7] and such a coloring can be computed efficiently [6].

3 Bipartite Speed Dating

In this section we design an algorithm for the bipartite case of SPEEDDATING. The key observation here is that any edge set E' satisfying the upper bound on the number of dates for each vertex forms a bipartite graph with maximum degree at most σ, and therefore it can always be colored with σ colors. Hence, the coloring subproblem is trivially solvable and does not impose any additional constraints on the subgraph that needs to be selected. This simplifies the problem to finding a subgraph of G that maximizes the total weight among all solutions that minimize the fairness violation. Our algorithm therefore works in three phases.

1. Determine the minimum value δ such that an edge set E' with $\ell(v) - \delta \leq \text{dates}(v) \leq h(v)$ for all $v \in V$ exists, using a binary search.
2. Determine the maximum weight edge set E' of G with $\ell(v) - \delta \leq \text{dates}(v) \leq h(v)$ for all $v \in V$ for the fixed value of δ determined in the previous phase.
3. Color the edge set determined in the second phase with at most $\Delta_{G'}$ colors.

We will now describe the phases. Note that due to the properties of the bipartite edge coloring problem, Phase 3 is completely independent from the previous two phases. This independence is however not given between Phases 1 and 2. In fact, Phase 1 will make use of the algorithm developed in Phase 2 to check whether a solution exists for a given fixed δ.

Phase 1: Determining the minimum fairness violation. Given a fixed fairness violation δ, the speed dating problem can be transformed into an equivalent instance, where the lower bounds on the number of dates are strict, i.e., $\ell(v) \leq \text{dates}(v)$ must hold for each node v by setting $\ell(v) \leftarrow \max\{0, \ell(v) - \delta\}$. After this transformation, finding an uncolored edge set boils down to the *weighted degree-constrained edge-subset problem* (WDCES).

Problem 1 (WDCES). Given a graph $G = (V, E)$ with edge weights $w : E \longrightarrow \mathbb{N} \setminus \{0\}$, lower and upper node capacities $\ell(v), h(v) : V \longrightarrow \mathbb{N}$ with $\ell(v) \leq h(v) \leq \deg(v)$, determine an edge set $E' \subseteq E$ of maximum weight such that $\ell(v) \leq \text{dates}(v) \leq h(v)$ holds for all nodes.

This problem is also known as weighted b-matching with unit edge capacities and has been previously studied by Gabow [3]. In the description of Phase 2 we will describe a simple MINCOSTFLOW based algorithm for solving this problem in the bipartite case. In Phase 1 we essentially discard the weights and simply wish to determine whether any feasible solution exists. We use this to determine the minimum value of the fairness violation δ by a binary search. The optimum value of δ is in $\{0, \ldots, \max h(v)\}$. If, for a fixed δ, the resulting WDCES problem does not admit a feasible solution, then smaller values of δ are also infeasible. Similarly, feasible solutions remain feasible for higher values of δ. Hence, binary search can be used to determine the smallest value δ for which the instance becomes feasible. This takes $O(\log n)$ feasibility tests.

Phase 2: Determining a Maximum Weight Subgraph. This phase consists of solving the WDCES problem that is obtained from an instance of bipartite SPEEDDATING by fixing the fairness violation to a certain value δ. To this end, we model the problem as a min cost flow problem (MINCOSTFLOW) as follows.

Let $G = (V,E)$ be an instance of WDCES with weight function w and lower and upper bounds for the nodes ℓ and h, stemming from fixing the fairness violation of a bipartite speed dating instance to a certain value δ. We construct an instance of min cost flow as follows. Recall that G is bipartite and hence $V = V_1 \cup V_2$ with $V_1 \cap V_2 = \emptyset$ and all edges in E have endpoints in both sets. The flow network N has vertices $V_1 \cup V_2 \cup \{s,t\}$, where s will act as a super source and t as a super sink. For each edge $uv \in E$ with $u \in V_1$ and $v \in V_2$ we add the arc (u,v) to N and set its lower capacity $\ell_N(u,v) = 0$ and its upper capacity $h_N(u,v) = 1$, and its cost $c(u,v) = -q(uv)$. We call these arcs *main arcs*, all other arcs will be *helper arcs*. For each vertex $u \in V_1$ we add an arc (s,u) with lower capacity $\ell_N(s,u) = \ell(u)$ and upper capacity $h_N(s,u) = h(u)$. Similarly, for all vertices $v \in V_2$ we add arcs (v,t) again with lower capacity $\ell_N(v,t) = \ell(v)$ and upper capacity $h_N(v,t) = h(v)$. All helper arcs have cost 0.

We set the desired flow value p that should flow from s to t to $|E|$. Further, we introduce an arc from s to t with $\ell_N(s,t) = 0$, $h_N(s,t) = p$ and $c(s,t) = 0$. This ensures that enough flow can be routed through the network to possibly select all edges in E, and on the other hand that superfluous flow can be piped along the (s,t)-arc with no cost. The transformed graph contains $n+2$ nodes and $n+m+1$ edges and thus its size is linear in the size of G. For a flow ϕ we denote its cost by $c(\phi) = \sum_{(u,v)\in N} \phi(u,v) \cdot c(u,v)$.

We show that solving the min cost flow problem yields an optimal solution to the WDCES problem, we omit the proof due to space constraints.

Lemma 1. *Let $G = (V_1 \cup V_2, E)$ be a bipartite instance of the WDCES problem and let N be the flow network constructed as above and let ϕ be an optimal solution of N that is integral. Then the edge set $E' = \{uv \in E \mid u \in V_1, v \in V_2, \phi(u,v) = 1\}$ is an optimal solution of the WDCES problem.*

Using an algorithm by Goldberg and Tarjan [5], this MINCOSTFLOW problem can be solved in time $O(nm\log^2 n)$ (note that the maximum absolute cost of our instance is 1), thus the running time for Phase 2 is in $O(nm\log^2 n)$. Further, in Phase 1 we do not require an optimal solution, instead we simply check whether a feasible solution exists using a MaxFlow algorithm with a running time of $O(nm\log n)$ [4]. The final step consists of edge-coloring the bipartite graph resulting from the previous phase, which requires $O(m\log n)$ time, using the algorithm by Cole and Hopcroft [2]. The following theorem summarizes our findings.

Theorem 2. *Let (G,q,ℓ,h,σ) be an instance of bipartite* SPEEDDATING *such that G has n vertices and m edges. An optimal solution can be computed in $O(nm\log^2 n)$ time.*

4 General Speed Dating

As we have seen in the previous section, the bipartite speed dating problem admits a polynomial-time algorithm. The crucial observation was that for bipartite graphs the third phase, consisting of coloring the graph with at most σ colors, can be carried out

independently from the result of the previous two phases. In the general case an optimal solution to the first two phases is an edge set E' that in general no longer induces a bipartite graph and therefore may not admit a coloring using only σ colors. Hence, even if we find a solution to the WDCES problem of Phase 2 (the above reduction to a flow problem relied on bipartiteness), the problem arising in the third phase is not necessarily solvable.

Suppose that G' is any subgraph determined in the first two phases. Although G' cannot necessarily be colored with σ colors and it is NP-complete to find out whether this is the case, by Vizing's theorem [7], we know that G' can be colored with at most $\sigma+1$ colors; we obtain a schedule that has one round too much. To remedy this, we introduce a new phase that simply removes the color class with the smallest total weight. In summary our algorithm works as follows.

1. Determine the minimum value δ such that an edge set E' with $\ell(v)-\delta \leq \text{dates}(v) \leq h(v)$ for all $v \in V$ exists, using a binary search.
2. Determine the maximum weight edge set E' of G with $\ell(v)-\delta \leq \text{dates}(v) \leq h(v)$ for all $v \in V$ for the fixed value of δ determined in the previous phase.
3. Color the edge set determined in the second phase with at most $\sigma+1$ colors.
4. If the previous phase uses $\sigma+1$ colors remove the edges of the lightest color.

Phase 1 works in the same way as in the bipartite case. Phase 2 can be reduced as in the bipartite case to WDCES. The reduction to MINCOSTFLOW however breaks, because the graph is not necessarily bipartite. To solve general WDCES we make use of an exact polynomial-time algorithm developed by Gabow [3], which runs in time $O(n^2\sigma^2\Delta^3 \log n)$. We avoid an additional factor of $\log n$ for Phase 1 by neglecting the weights during the binary search, which, using the same reduction, allows for a faster algorithm with running time $O(nm\Delta + n^2\Delta^2)$. Phase 3 can be solved using Vizing's algorithm with running time $O(nm)$ [6]. Phase 4 sums up the weights of the edges of each color and removes the edges of the lightest color if necessary, which requires $O(n+m)$ time. The total running time for the algorithm therefore is $O(n^2\sigma^2\Delta^3 \log n)$. We now show that the algorithm gives provable performance guarantees on the quality of the solutions.

If Phase 3 succeeds with coloring the resulting graph using at most σ colors, Phase 4 does not remove any edges and since the first two phases are solved optimally, the algorithm calculates an optimal solution in this case. For the sake of deriving worst case performance guarantees we therefore assume that this is not the case and that in Phase 4 all edges of the lightest color are removed. We have the following lemma.

Lemma 2. *Let (G,q,ℓ,h,σ) be an instance of* SPEEDDATING. *Let $G_{\text{OPT}} = (V, E_{\text{OPT}})$ be an optimal solution and let $G' = (V, E')$ be the solution computed by the above algorithm. Then $\delta(G') \leq \delta(G_{\text{OPT}})+1$ and $q(G') \leq \frac{\sigma}{\sigma+1} q(G_{\text{OPT}})$.*

Proof. Let E_0 be the edge set that was computed in Phase 2 of the algorithm during the execution that resulted in G'. We denote by G_0 the graph (V, E_0).

We first bound the fairness violation. Since E_0 is an optimal solution to Phases 1 and 2, we have $\delta(G_0) \leq \delta(G_{\text{OPT}})$. Moreover, since E' results from E_0 by removing a matching we have $\delta(G') \leq \delta(G_0)+1$. Together with the previous inequality this yields $\delta(G') \leq \delta(G_{\text{OPT}})+1$.

For the overall quality consider again that E_0 is an optimal solution to the first two phases and hence $q(G_0) \geq q(G_{\text{OPT}})$. Recall that G' results from G_0 by removing the edges of the lightest color, denote them by E_{light}. By the pigeon-hole principle we have $q(E_{\text{light}}) \leq q(E_0)/(\sigma+1)$. For the overall quality of G' we thus get $q(G') = q(G_0) - q(E_{\text{light}}) \geq q(G_0) - \frac{q(G_0)}{\sigma+1} = \frac{\sigma}{\sigma+1} q(G_0) \geq \frac{\sigma}{\sigma+1} q(G_{\text{OPT}})$. This concludes the proof of the lemma. □

The following theorem summarizes the results of this section.

Theorem 3. *Let (G,q,ℓ,h,σ) be an instance of* SPEEDDATING. *Let G_{OPT} be an optimal solution. A solution G' with $\delta(G') \leq \delta(G_{\text{OPT}})+1$ and $q(G') \geq \sigma/(\sigma+1) \cdot q(G_{\text{OPT}})$ can be computed in $O(n^2\sigma^2\Delta^3 \log n)$ time.*

Note that for realistic values of σ, e.g., $\sigma = 12$, we have $\sigma/(\sigma+1) \geq 0.92$, i.e., the solution quality is at least 92% of the optimum. While the algorithm is favorable in terms of fairness and quality, its worst-case time complexity is quite high. We therefore also propose a greedy algorithms for the general speed dating problem, which may be advantageous in terms of running time.

We note that the running times of the algorithm for the bipartite case as well as of the approximation algorithm can be improved to $O(n\sigma \min\{m \log n, n^2\})$ using a more sophisticated reduction by Gabow [3]. However, this algorithm is very complicated, as it requires to modify a weighted maximum matching algorithm to dynamically manipulate the graph it is working on during the execution. We therefore chose to present the running times that more closely match the complexity of the implementations we are going to evaluate.

A Greedy Strategy for General Speed Dating. Let (G,q,ℓ,h,σ) with $G=(V,E)$ be an instance of SPEEDDATING. The algorithm GREEDY maintains a set S of edges that are colored with σ colors. In the course of the algorithm edges are only added to S and never removed. Moreover, every edge is colored upon insertion with one of the σ colors such that the graph (V,S) is properly σ-edge-colored and no edge in S ever changes its color. To pick the next edge, the algorithm picks the edge uv with the highest valuation $k(uv)$, where k is some formula computing the value of an edge. We use $k(uv) = w_{\max} \max\{\delta_S(u), \delta_S(v)\} + q(uv)$, where $w_{\max} = \max_{e\in E} q(e)$, i.e., it priorizes edges that are incident to a vertex whose fairness constraints are strongly violated in the current solution, the quality is a second criterion. The algorithm iteratively finds the edge with the highest value $k(uv)$ and either adds it to S, if this is possible, i.e., if $\text{dates}_S(u) \leq h(u)-1$ and $\text{dates}_S(v) \leq h(v)-1$ and the endpoints u and v have a common unused color, which then becomes the color of uv. If uv cannot be added, it is discarded and removed from E. The algorithm stops when $E = \emptyset$. The algorithm can be implemented to run in $O(m\Delta \log m)$ time.

5 Evaluation

We implemented the algorithms described above in C++. Our implementations use the LEMON 1.2 library[1], which provides efficient implementations of MINCOSTFLOW and weighted maximum matching algorithms. All our experiments were run on one core of a computer with an Intel Q6600 processor with 4MB cache and 2GB RAM. As compiler we used g++ version 4.4.1 with compiler flags `-DNDEBUG` and `-O3`.

Problem Instances. We evaluate our algorithms on a variety of problem instances. We use four types of instances. The first two are produced by random generators that produce randomly filled-out forms and build graphs based on the similarity of these forms. Additionally, we evaluate our algorithms on social networks. The reason for this is that instances to SPEEDDATING are graphs that model the probability that people like each other. It is to be expected that such graphs do not strongly deviate from graphs modeling that people actually know or like each other, i.e., social networks. We therefore use the small world generator by Brandes et al. [1] to produce instances. Additionally, we use twelve instances of a real-world social network, stemming from the email network of the Faculty of Informatics at the Karlsruhe Institute of Technology.

In all cases, we set the number of rounds σ to 12. Assuming that each date lasts ten minutes, this implies that the whole event lasts at least two hours, which appears to be a realistic total duration. In the general case we always set $\ell(p) = h(p) = \min\{12, \deg(p)\}$ for all persons p in the instance. For the bipartite case we choose the genders randomly with a change of 40% for being a woman. Let N be the number of men and M the number of women. We set $\ell(w) = h(w) = \min\{12, \deg(w)\}$ for all women w. For all men m we set $\ell(m) = \lfloor \sigma M/N \rfloor$ and $h(m) = \lceil \sigma M/N \rceil$ to evenly distribute the dates.

For the generation of the edges and their weights, recall that each participant fills out a form about himself and his ideal partner before the speed dating event. Our random instance generators are based on randomly generating the contents of such forms and then computing for each possible pairing of two persons x and y a *score* for the edge xy, based on their forms. If the score of an edge xy is non-positive, we discard the pairing so that the resulting graph is not necessarily complete. For the bipartite case, we restrict the considered pairings to pairs of a man and a woman.

The first generator models the forms using random vectors of size 30, each entry drawn uniformly at random from $\{0/5, \ldots, 5/5\}$. The score of a pair xy is based on the Euclidean distance of their corresponding vectors v_x and v_y. We compute the weight of the edge xy as $\lfloor (\sqrt{30}/2 - |v_x - v_y|_2) \rfloor$ and discard it, if it is not positive. Note that the weight is positive about 50% of the time.

The second generator tries to model more complex forms. For each person we generated a random form composed of 30 questions about what traits the ideal partner should have. For each question q every person p answers on a scale of 0 to 5 how well she fulfills these traits ($\text{has}_q(p)$) and how the partner should fulfill it ($\text{wants}_q(p)$). Further each person must indicate for each question the level of importance on a scale of 0 to 5 ($\text{imp}_q(p)$). Again, we assume that all choices are uniformly distributed.

[1] Available at `http://lemon.cs.elte.hu/trac/lemon`

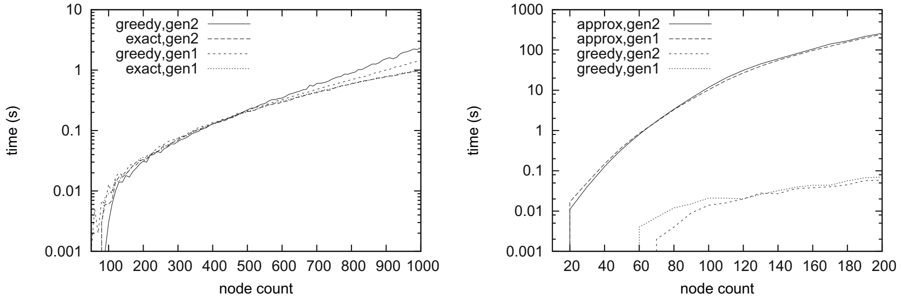


Fig. 1. Comparison of the running times of the exact algorithm and the greedy algorithm for generators 1 and 2 on bipartite instance (left) and comparison of the running times for the approximation algorithm and the greedy heuristic on instances of the general speed dating problem (right).

We define a function that estimates whether x likes y as $\text{likes}(x,y) = \alpha \cdot \sum_{q=1}^{30} \text{imp}_q(x) - \sum_{q=1}^{30} \left|\text{has}_q(y) - \text{wants}_q(x)\right| \cdot \text{imp}_q(x)$. The first term acts as a threshold telling how demanding the person x is. The second term evaluates how close y is to the profile x would like to have for his partner, weighted by the importance that x gives to each of the traits. Since the score function we use should be symmetric, we set $\text{score}(xy) = \text{likes}(x,y) + \text{likes}(y,x)$.

The constant α is a tuning parameter that affects the density of the resulting graph. For $\alpha = 2$ the generator produces almost complete graphs, for $\alpha = 3$ the graphs are extremely sparse. We choose $\alpha = 2.3$ for our experiments, as this generates plausible, relatively dense graphs.

As instances from social networks, we use graphs generated according to the small world model, generated by the generator of Brandes et al. [1]. The parameters we use for the generator are the following. We generate instances with 200 nodes and set the rewiring probability $p = 0.2$. The minimum degree k ranges from 10 to 100 in steps of 2. For each value of k we use five random samples to even out random influences. The edge weights are integers, chosen uniformly at random in the range from 1 to 20.

Finally, we use real-world social networks stemming from the email network of the Faculty of Informatics at the Karlsruhe Institute of Technology. For each month from September 2006 to August 2007 the corresponding graph contains all people of the network and every pair is connected by an edge whose weight is the number of emails they exchanged.

Experiments. We first present the results on instances produced by our form based generators. In our plots we indicate which generator was used by either gen_1 or gen_2. For the bipartite speed dating problem, we generate instances with n persons, where n ranges from 100 to 1000 in steps of size 10. The approximation algorithm is much slower, due to the different reduction that is necessary in Phase 2. We therefore take only instances of sizes 10 up to 200, again in steps of size 10. To remedy outliers, we average the results over ten samples of each size in both cases. We also run the greedy algorithm on all the instances.

Figure 1 shows the running times of our algorithms both in the bipartite and in the general case. Surprisingly, in the bipartite case, the greedy algorithm is constantly

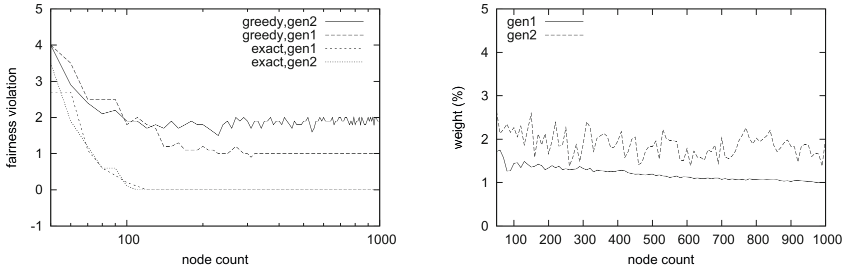


Fig. 2. Solution quality of the solutions produced by the exact solver and the greedy algorithm for bipartite instances generated by gen_1 and gen_2

slower than our exact solution algorithm although it has a far better worst case complexity. This is probably due to the fact that we did not take a lot of care of micro-optimizing the inner loops of the greedy algorithm whereas the inner working of the MINCOSTFLOW solver provided in the LEMON library are carefully tuned. All four curves suggest a quadratic growth in the number of people. As there are quadratically many potential dates this is linear in the size of the input.

For the general case, the situation is quite different. While for moderate sizes of around 100 people, an approximate solution can still be computed in less than two minutes, this quickly grows to roughly 10 minutes for events with 200 people. Expectedly, the greedy algorithm grossly outperforms the approximation algorithm in terms of running time and solves all instances within less than 0.1 seconds. As we will see later, the running time of the approximation algorithm is much better for sparser instances. Nevertheless, even for dense graphs the approximation algorithm needs less than ten minutes in the worse case tested, which is still within the acceptable bounds given by the problem motivation.

Next, we compare the solution quality of our algorithms. We plot the δ values of all solutions. Since the absolute quality value does not really have a means of interpretation, we only show the relative difference of the greedy solution and the solution computed by the exact and the approximate algorithm, respectively. A value of 3% indicates that the weight of the solution computed by the greedy algorithm is 3% higher than the solution computed by the exact algorithm (for bipartite instances) or the approximation algorithm (for general instances). A negative percentage indicates that the weight of the solution computed by the greedy algorithm is smaller.

Fig. 2 shows the performance of the greedy algorithm on bipartite instances with respect to quality. For the fairness violation δ, note that except for the very small instances, the greedy algorithm misses the optimal δ value usually by 1 or 2. Considering that there are only 12 rounds and that minimizing the fairness violation is our main optimization criterion, being off by 2 units is rather bad. In terms of overall quality, the greedy algorithm performs quite well and is never off the true value by more than 5%, which is acceptable as it is unlikely that participants would notice this.

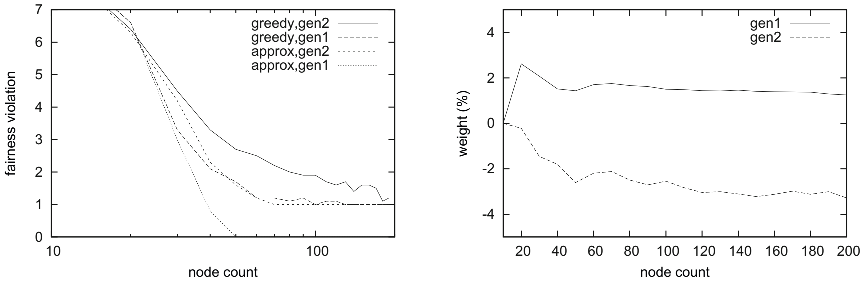


Fig. 3. Solution quality of the solutions produced by the approximation algorithm and the greedy algorithm for general instances of both generators

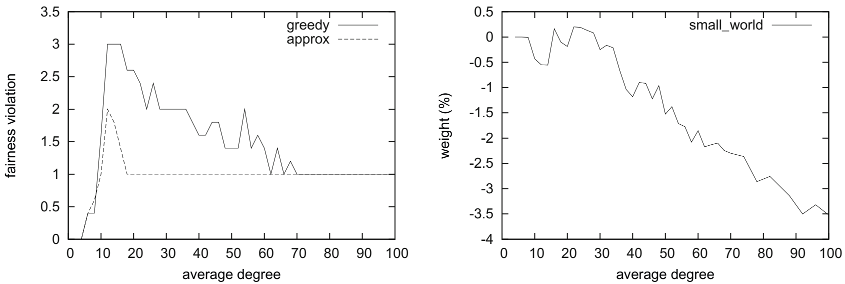


Fig. 4. Comparison of the greedy and approximation algorithm on small world graphs of varying density

Fig. 3 shows the corresponding evaluation for the general case. The greedy algorithm performs quite well, for instances of the first generator it misses the optimal value by only 1 for smaller instances and finds an optimal value for larger instances. For the second generator, the approximation algorithm does not find optimal solutions and starting from a certain size, always finds solutions with fairness violation 1. Since this fairness violation stems from discarding some edges and the graphs are rather dense, it is not unlikely that the optimal fairness violation may be 0. The greedy algorithm performs comparably, is slightly worse for small instances and achieves the same fairness violation as the approximation for most larger instance sizes. Interestingly, the greedy algorithm outperforms the approximation algorithm for instances from gen_2 in terms of quality by up to 5%.

Figure 4 shows the results of the experiments on small world graphs. Note that for this experiment the number of vertices is fixed to 200 and the average degree varies from 4 to 100. The approximation algorithm performs consistently better than the greedy algorithm. Only for very dense graph does the greedy algorithm find solutions with the same δ but with less weight. Moreover, the running time of the approximation algorithm is consistently below one second. Finally, the results on the email networks are shown in Table 1. Again the approximation algorithm is favorable in terms of δ but slightly worse than greedy in terms of total quality. Moreover, this shows that the approximation algorithm is extremely fast on sparse instances.

Table 1. Performance of our algorithms on instances stemming from the email network at the Karlsruhe Institute of Technology. For time and δ, the first value is for the approximation algorithm, the second for the greedy algorithm. The quality column shows the difference between the greedy and the approximate solution, relative to the approximate solution quality.

month	n	m	time [s]		δ		q [%]
1	59	2994	1.86	0.12	3	4	+2.1
2	431	1898	0.37	0.04	2	5	+2.8
3	444	1729	0.31	0.04	2	3	+3.7
4	432	1660	0.32	0.04	2	4	+5.1
5	450	1846	0.43	0.06	3	3	+0.2
6	430	1815	0.44	0.03	3	4	+0.9
7	440	1906	0.47	0.06	3	4	+2.1
8	405	1563	0.24	0.05	2	3	+2.8
9	432	1633	0.28	0.04	2	3	+0.7
10	558	2180	0.67	0.07	3	4	+0.2
11	504	1994	0.42	0.06	2	4	+5.5
12	881	3427	4.71	0.16	3	5	+7.1

Conclusion. Our experimental evaluation shows that for the bipartite speed dating problem, our exact polynomial-time algorithm is the preferred solution. It produces exact results, is very fast and solves even problem instances with 1000 people within a few seconds and therefore is the method of choice for these instances. For the general speed dating problem the situation is not that clear. The approximation algorithm takes a few minutes to find solutions for large, dense instances and the quality of solutions found by the greedy algorithm is in many cases comparable in this case. To maintain the theoretical guarantees the best solution for this case seems to be to run both algorithms and to pick the better solution. If execution time is crucial, the greedy algorithm is the algorithm of choice for these instances as it is very fast and likely to produce solutions of high quality. For sparser instances the approximation algorithm is both fast and gives better results than greedy.

Acknowledgments. We thank Robert Görke for helpful discussions and providing us with the email networks we used for evaluation.

References

1. Batagelj, V., Brandes, U.: Efficient generation of large random networks. Physical Review E 036113 (2005)
2. Cole, R., Hopcroft, J.: On edge coloring bipartite graphs. SIAM J. Comput. 11(3), 540–546 (1982)
3. Gabow, H.N.: An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In: Proc. 15th Annu. ACM Sympos. Theor. Comput(STOC 1983), pp. 448–456. ACM, New York (1983)
4. Goldberg, A., Tarjan, R.E.: A new approach to the maximum flow problem. J. Assoc. Comput. Mach. 35, 921–940 (1988)
5. Goldberg, A., Tarjan, R.E.: Finding minimum-cost circulations by canceling negative cycles. J. Assoc. Comput. Mach. 36, 873–886 (1989)
6. Misra, J., Gries, D.: A constructive proof of Vizing's Theorem. In: Inf. Proc. Let., pp. 131–133 (1992)
7. Vizing, V.G.: On an estimate of the chromatic class of a p-graph. Diskret. Analiz, pp. 25–30 (1964) (in Russian)

Experimental Evaluation of Algorithms for the Orthogonal Milling Problem with Turn Costs*

Igor R. de Assis and Cid C. de Souza

Institute of Computing, University of Campinas (UNICAMP), Brazil
igor.assis@gmail.com, cid@ic.unicamp.br

Abstract. This paper studies the Orthogonal Milling with Turn Costs. An exact algorithm is proposed based on an Integer Programming formulation of the problem. To our knowledge, this is the first exact algorithm ever proposed for the problem. Besides, a simple heuristic is also presented and an unprecedented experimentation involving these two algorithms and an existing approximation algorithm is carried out. We report and analyze the results obtained in these tests. Our benchmark instances are made public to allow for future comparisons.

Keywords: orthogonal milling with turn costs, exact algorithms, approximation algorithms, heuristics, experimental evaluation.

1 Introduction

In this paper we make an experimental evaluation of algorithms for the *orthogonal discrete milling problem* (ODMP). Before giving a formal definition of ODMP, we describe a simple application of it.

Consider a garden in which a gardener has to mow the lawn. Assume that the area of the garden is defined by an orthogonal polygon and has some reflection pools inside whose forms are also given by orthogonal polygons. Suppose a lawn mower is available whose area is defined by a square of size ℓ. The mower can only move parallel to the axis of the edges of the polygon that limits the garden. Moreover, assume that the length of all edges of the orthogonal polygons are positive integer multiples of ℓ.

To accomplish his task, the gardener needs to find a path along which to move the lawn mower in order that the sweep of the mower covers the entire region, removing all the lawn, but never going outside the boundaries of the garden or through the reflecting pools. Though moving in a straight line is easy, a turn is a more laborious operation. So, it is natural for the gardener to look for a path that minimizes the number of right turns (notice that in this context, a U-turn is considered as being two right turns).

* This research was partially supported by CNPQ (Conselho Nacional de Desenvolvimento Científico e Tecnológico) – grants #301732/2007-8, #473867/2010-9, FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) – grant #07/52015-0, and a grant from FAEPEX/UNICAMP.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 304–314, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Applications of this problem (or variants of it) abound. According to [1], they can be found in many fields, including numerically controlled (NC) machining applications, automatic inspection, spray painting/coating operations, robotic exploration and arc routing.

Later in this paper we will see that the ODMP can be modeled as a graph optimization problem. In fact, it belongs to a broader class of problems entitled *"Minimum-Turn Cycle Cover in Planar Grid Graphs"*, or MTCC for short, and listed as the 53rd problem in the list of *"The Open Problems Project"* [2]. There it is said that the motivation to study such problems is that "minimizing turns is a natural geometric measure; understanding its algorithmic behavior is of general interest."

The present paper aims to contribute in this vein. To this end, we develop both exact and heuristic algorithms for the ODMP. Experiments with the implementations of the algorithms are conducted to assess the efficiency of both algorithms. We also implemented the best known approximation algorithm available to date [3] for the ODMP and compare the three methods with respect to the values of the solutions they obtain.

Our contributions. Despite the relevance of the ODMP and the recognition of the need to have more algorithmic insights about the problem [2], to the best of our knowledge, this is the first time an exact algorithm is proposed to solve it. Besides, no heuristics have been proposed so far and no public benchmark exists with instances allowing researchers to compare their algorithms. This paper helps filling all these gaps and is the starting point for future developments, especially those related to exact solutions based on integer programming models and to heuristics.

Organization of the text. The ODMP is formalized in the next section, where we briefly review the literature. In section 3 we discuss how to formulate the ODMP as an integer programming (IP) model and how to compute such model. In section 4 we give the main steps of the approximation algorithm we used in our experiments, while section 5 is devoted to the presentation of a simple heuristic we developed for the ODMP. The computational tests and the analysis of the results obtained are the topic of section 6. Finally, in section 7, we draw some conclusions and discuss future research directions.

2 Problem Description and Literature Overview

Consider the lawn mowing problem defined in the previous section. Let P denote the (bounded) orthogonal polygon representing the garden and Q be the unit square corresponding to the lawn mower or cutter. Moreover, let $\mathcal{H} = \{H_1, H_2, \ldots, H_p\}$ be the set of orthogonal polygons associated to the reflection pools and named *holes*. The region determined by $P \setminus \mathcal{H}$ is called the *pocket*. For simplicity, assume that the left/uppermost vertices of P and Q coincide.

A feasible solution for ODMP is a closed curve, not necessarily simple, traversed by Q whose Minkowski cover is equivalent to the pocket. The curve is limited to

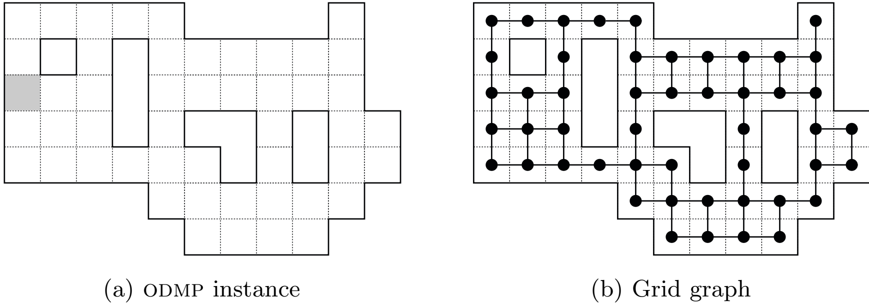


Fig. 1. Example instance of ODMP and the corresponding grid graph

have axis-parallel edges and, hence, all its turns are either orthogonal (90°) or U-turns (180°). Orthogonal turns have cost 1 while U-turns are assigned with cost 2.

Under the conditions described above, the region can be considered to be the (connected) union of n *pixels*, i.e., axis-parallel unit squares with vertices having integer coordinates. In this case, a feasible solution can be viewed as a curve where the turns occur at points with coordinates of the form $(k+1)/2$ for some integer k.

Now, let V be the set of points of the pocket with such coordinates. These points correspond to the centroids of all pixels and, by placing the cutter in one such point, the associated pixel is covered. Define E to be the set of all axis-parallel segments joining pairs of points in V and not intersecting other points of this set. Then, $G = (V, E)$ is a grid graph, named the *subdivision graph*, and a feasible solution to ODMP can be cast as *Hamiltonian walk* of G. In this walk, one can add a cost of 1 each time the walk goes from a "vertical" edge to a "horizontal" one or vice versa, a cost of 2 if the same edge is traversed in a row, and, otherwise, no cost is incurred if subsequent edges are both in the same direction. The total cost of the walk is the sum of all these costs. It is clear that to solve the ODMP one has to find the cheapest walk in G.

Figure 1 illustrates these concepts. In (a) it is depicted an instance of ODMP. The cutter Q, identified by the gray square, and the squares having dashed sides comprise the set of all pixels which forms a planar subdivision of the pocket. The corresponding grid graph G is shown Figure 1 (b).

In [1] the ODMP is shown to be $\mathcal{NP}$-hard. That paper also contains several references on theoretical results known for several sorts of MTCC problems and is an excellent starting point for those interested in this topic. Among the relevant references for the ODMP, we highlight those related to arc routing problems since some of them are akin to the exact approach proposed here.

Many exact and heuristic algorithms for arc routing problems with turn costs are based on reductions of the original problem to some variant of the Symmetric or Asymmetric Traveling Salesman Problem (STSP or ATSP, cf [4]). For the purpose of the discussion that follows, let G' be the graph obtained from one such transformation.

Typically, the resulting routing problem requires the computation of an optimal cycle in G', which may be constrained to be Hamiltonian or to visit a given subset of vertices (cf [5]). The modeling we describe in section 3 differ from these approaches in that each vertex of G' belongs to at least two predefined subsets of vertices and the cycle that is sought has to visit at least one vertex in each of these subsets. The next section is devoted to the development of an IP model for ODMP resulting in the design of a branch-and-cut algorithm to solve it exactly.

3 An Exact Algorithm for the ODMP

Let $G = (V, E)$ be a subdivision graph corresponding to an instance of ODMP, where $|V| = n$. We define $G' = (V', A)$ as the directed graph where, for each edge $e = \{i, j\}$ of G, there are two vertices ij and ji in V', one for each orientation of e. Besides, there is an arc between two vertices ij and xy in G' if and only if $j = x$, meaning that an arc in G' represents a turn in G. We call G' the *turn graph* of G.

Now, for $i \in V$, let C_i be the set of vertices of G' associated with edges of G incident in i, i.e., the set C_i contains all vertices in V' of the form xi and ix. The sets C_i are called the *clusters* of G'. Finally, we assume that, for each arc $(u, v) \in A$, it is assigned a cost c_{uv} relative to the turn formed by the two corresponding edges in G.

It is easy to see that a milling tour of G corresponds to a closed walk in G' that visits each cluster C_u at least once. Hence, finding a minimum cost milling tour in G is equivalent to find one such closed walk in G' having minimum cost.

With the definitions above, we are able to describe the integer programming model for the ODMP in terms of G'. For each arc (u, v) of G', the integer variables x_{uv} count the number of times the arc is traversed in the solution and, for each vertex $u \in V'$, the binary variable y_u is set to one if and only if the vertex u is in the solution. Thus, the model is given by:

$$\min \sum_{(u,v)\in A} c_{uv} x_{uv}$$

$$\text{s.t} \quad \sum_{(u,v)\in A} x_{uv} = \sum_{(v,u)\in A} x_{vu} \quad \forall\, u \in V', \tag{1}$$

$$\sum_{u \in C_i} y_u \geq 1 \quad \forall\, i \in V, \tag{2}$$

$$\sum_{u\in U, v\notin U} x_{uv} \geq y_w \quad \forall\, U \subset V' : \exists\, i \in V : U \cap C_i = \emptyset,\ w \in U, \tag{3}$$

$$x_{uv} \leq n y_u \quad \forall\, u \in V', \forall\, (u, v) \in A, \tag{4}$$

$$x_{uv} \in \mathbb{Z},\ y_w \in \mathbb{B} \quad \forall\, (u, v) \in A,\ w \in V'.$$

By restriction (1) the number of times the tour enters a vertex is equal to the number of times it leaves that same vertex. Constraint (2) says that each set C_u

must be visited at least once by the tour. Constraints (3) are needed to avoid disjoint subtours. Constraint (4) limits the number of times an arc can be in a solution. Since in any optimal walk an arc (u, v) is traversed only if the cutter is moving towards an unvisited cluster and there are exactly n clusters, we impose that an arc is traversed at most n times. We are still investigating if we can reduce this upper bound. In any case, in our experiments no arc was traversed more than once in an optimal solution.

The number of constraints of type (3), called *subtour elimination*, is exponential in n. Thus, except for very small values of n, it is not practical to add them all to the formulation *a priori*. Instead, we use them as cutting planes. To do so, the associated *separation problem* has to be solved. The standard technique in such cases involves the computation of minimum cuts in graphs. Let $(x^\star, y^\star)$ be a solution of the relaxed problem and $S = \{u : y_u > 0\}$. This solution violates a subtour elimination constraint if and only if, for some $i \in V$ and $w \in S$, there exists $U \subseteq V' \setminus C_i$ such that $\sum_{u \in U, v \notin U} x^\star_{uv} < y^\star_w$. Thus, to decide whether or not there exists a violated subtour elimination relative to some choice of i and w, one has to find a minimum cut separating w from the cluster C_i, with arc weights computed from the values of the variables of the optimal solution of the linear relaxation. Therefore, a subtour elimination constraint violated by $(x^\star, y^\star)$ can be found polynomial time by solving one minimum cut problem for each pair (w, C_i) where $w \in S, i \in V$. Due to the equivalence between separation and optimization [6], this ensures that the optima of the linear relaxations at each node of the enumeration tree can be computed in polynomial time.

From the discussion above, it is immediate to devise a branch-and-cut algorithm for the ODMP. Experiments with this exact algorithm are reported in section 6.

As a final remark, we just observe that the reader familiar with IP formulations for arc routing problems may have noticed that our model for the ODMP resembles those for the *generalized* ATSP, denoted here by GATSP (see [7] and references cited there). However, a closer inspection of the two problems shows that they differ in some crucial aspects, among which, we highlight the following two: *(i)* in the ODMP the clusters (subsets of vertices) do not form a partition of the vertex set; and *(ii)* a feasible solution (tour) of the ODMP is allowed to visit a vertex more than once.

4 An Approximation Algorithm for the ODMP

In this section we give a summary of the approximate algorithm proposed in [3] for the ODMP and denoted here by `APX`. Prior to that, a few definitions are required.

A maximal vertical or horizontal segment joining the centroids of two pixels is called a *strip* if the area that is covered when the cutter walks along this segment is entirely inside the pocket. A *strip cover* is a set of strips such that the union of their Minkowski covers (relative to the cutter) is equal to the pocket. A *rook* placed in a pixel p is said to *attack* a pixel p' if there exists a horizontal or vertical line segment in the pocket having the centroids of these pixel as extremities. A

rook placement is a set of rooks placed such that no two rooks attack each other. With these definitions, we are ready to state a fundamental result supporting the approximation algorithm.

Theorem 1. *For orthogonal discrete milling a minimum-sized strip cover and a maximum-sized rook placement have equal size.*

Proof. Let $B = (V_1, V_2, E)$ be a bipartite graph where V_1 are the horizontal strips and V_2 the vertical ones. There is an edge $\{v_1, v_2\} \in E$ for $v_1 \in V_1$ and $v_2 \in V_2$ if the strips have a pixel in common. It is easily seen that a maximum matching M in B corresponds to a maximum rook placement and a minimum vertex cover K corresponds to a minimum strip cover. Hence, by the König-Egerváry theorem (cf, [8]) M and K have the same cardinality. □

The first step of the approximate algorithm is to find a minimum strip cover which, from Theorem 1, can be done in polynomial time in the number of pixels (n) by a bipartite matching algorithm (cf, [8]). The next step involves connecting the strips of the minimum strip cover to form a cycle cover of the subdivision graph G. To do that, we construct an auxiliary complete graph $H = (V_H, E_H)$ where V_H is the set of endpoints of the strips in the strip cover. It is worth noting that we use different copies of endpoints representing the same subregion. The weight of an edge $\{u, v\}$ of E_H is the length of the shortest path from u to v measured relative to turn costs. The following two steps of the approximation algorithm involve the computation of a minimum weight perfect matching in H, and the subsequent construction of the cycle cover by joining the strips with paths represented by the edges of the matching. It is important to note that when computing the shortest path we must consider the direction, vertical or horizontal, where the path starts: if it is the same of the strip of which the vertex is an endpoint we add one to the cost and zero otherwise.

We note that a simple way to find the shortest path between two pixels u and v is to use the turn graph. Let $G' = (V', E')$ be the turn graph of a subdivision graph $G = (V, E)$ and $C_i \; \forall \; i \in V$ be the clusters of G'. The cost, with respect to turns, of the shortest path between $u, v \in V$ is the cost of the shortest path between C_u and C_v.

The final step of the algorithm merges the cycles to find a complete milling tour. Merging cycles can be done in a way that adds at most two turns per merge. We say that two cycles *intersect* if they have a common pixel and are *adjacent* if they do not intersect and have at least two adjacent pixels. Let $T_1 = (u_0, \ldots, u_r)$ and $T_2 = (v_0, \ldots, v_s)$ be two cycles that intersect, $u_i = v_j$ a common pixel, u_{i-1}, u_{i+1} the neighbors in T_1, v_{j-1}, v_{j+1} the neighbors in T_2. The cycle $T = (u_0, \ldots, u_{i-1}, v_j, v_{j+1}, \ldots, v_s, v_0, \ldots, v_{j-1}, u_i, u_{i+1}, \ldots, u_r)$ cover every pixel of T_1 and T_2 and adds at most two turns.

Consider now the case of two adjacent cycles $T_1 = (u_0, \ldots, u_r)$ and $T_2 = (v_0, \ldots, v_s)$. Let u_i and v_j be adjacent pixels and, without loss of generality, assume that v_j is the leftmost pixel and is below u_i. This situation can be reduced to that of intersecting cycles. For this, we extend the cycle T_2 to include u_i to obtain the new cycle $T_2' = (v_0, \ldots, v_{j-1}, v_j, u_i, v_j, v_{j+1}, v_s)$. Now T_1 and

T_2' intersect at u_i and we can repeat the same operation as before to generate a cycle T covering T_1 and T_2 and having at most two extra turns.

Using the cycle merging procedures above we can convert a cycle cover with c cycles and t turns in a milling tour with $t+2(c-1)$ turns. The resulting algorithm has an approximation factor of 3.75. The proof of this and other related results, along with approximation algorithms for several variations of the general discrete milling problem, can be found in [3].

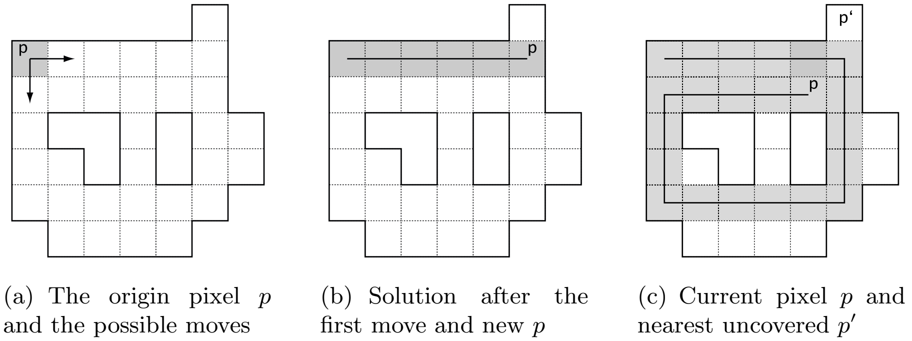
5 An Heuristic for ODMP

Another common approach when solving $\mathcal{NP}$-hard problems is to give up on theoretical performance guarantees and try to design algorithms that produce good results in practice, the so-called heuristics. In this section we present a simple but effective heuristic `HEUR` for the ODMP. We call it the *Gardener's algorithm* since it somehow mimics the human greedy strategy to tackle the problem.

The input of our heuristic is the subdivision polygon and one of its a pixel that is located at the boundary of the pocket representing the position where the cutter starts, named the *origin*. Generically, let us denote by p the pixel where the cutter is located at the beginning of an iteration of the algorithm. From this pixel, we calculate the movement of the cutter which visits the largest number of uncovered pixels (there are at most four possible movements, two for each direction). We execute this movement always ending at a previously uncovered pixel. If there is no such movement, i.e., they all lead to already covered pixels, then the cutter moves from p to the nearest (relative to turn costs) uncovered pixel or to the origin, in case all the pixels have been covered.

Figure 2 shows some steps of the execution of `HEUR` in an instance. In `(a)` is depicted the pixel p, corresponding to the cutter's initial location, and the two possible moves. The algorithm moves the cutter rightward because the number of uncovered pixels is five compared to four if the cutter goes downward. In `(b)`, it is indicated the partial tour after the first movement has been completed and the pixel p representing the new position of the cutter. Figure 2 `(c)` shows the situation after the fifth movement of the cutter. It illustrates the case where there is no horizontal or vertical movement that can reach an uncovered pixel. When this happens, the cutter's next move goes from its current location to the nearest uncovered pixel (p') which, in this example, corresponds to a path of cost one.

To improve the performance of the heuristic we borrowed some techniques of the GRASP metaheuristic [9], namely, multi-start and randomization. In our implementation, the multi-start strategy is enforced by fixing the number of iterations. As for the randomization of the algorithm, it was introduced in two steps: the choice of the movement at each iteration and the choice of the pixel to go when no movement is able to reach an uncovered pixel. In both situations, the selection is made from a restricted list of elements. In the first case, the size of the list is limited to two and the elements are sorted in monotonically decreasing order of uncovered pixels. In the second case, the list contains the $\lceil 0.01n \rceil$ nearest uncovered pixels, where n is the total number of regions. The randomization is based on the uniform distribution.



(a) The origin pixel p and the possible moves

(b) Solution after the first move and new p

(c) Current pixel p and nearest uncovered p'

Fig. 2. Examples of movements of the cutter in HEUR

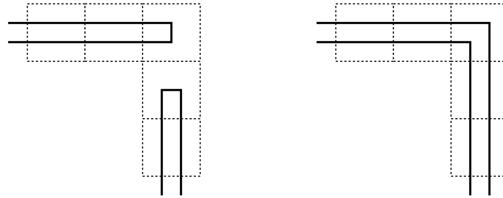


Fig. 3. Example of orthogonal U-turns that can be joined to reduce the cost in two

It is important to note that HEUR does not contain a local-search step as it is the case in the standard GRASP metaheuristic. This issue is currently being investigated. The difficulty is to find a suitable perturbation of the current solution in order to define a neighborhood that can be explored efficiently. For the ODMP, a generic perturbation can be thought as an operation that breaks the current tour, by removing some of its edges, and subsequently obtains a new tour by joining the resulting pieces via the addition of other edges. Ideally, the cost of the new solution should be smaller than that of the starting one. There are simple cases where this can be done easily, for instance, when there are orthogonal U-turns in adjacent pixels, as shown in figure 3. Despite its simplicity, the perturbation suggested by this example is not appropriate for the definition of a neighborhood for the problem. This is so because adjacent U-turns do not appear very often in reasonably good solutions of ODMP instances. Most importantly, the set of solutions that can be generated by repeatedly applying this operation is very limited and may not contain an optimal one.

6 Experimental Results

The algorithms described in the previous sections were tested on instances generated randomly according to the following procedure. Initially a square with side of (integer) length s is generated together with its subdivision graph $G = (V, E)$.

Then, two routines are applied in a row to generate the instance. In the first one, a random spanning tree of G is found and the edges of the tree are marked while all the remaining edges in G are set as unmarked. We now iterate once over each unmarked edge removing it from G with probability α if it is not incident to a vertex of degree two, in which case the edge becomes marked. The reason to discard the presence of vertices of degree one is that any feasible solution would be forced to make a U-turn at this vertex. Besides, it is easy to see that any instance having a degree one vertex can be transformed into an equivalent instance whose vertices have degree at least 2. Notice that, to this point, the number of vertices in G remains unchanged. Next we describe the second routine that allows us to eliminate vertices from G.

Given the subdivision graph $G = (V, E)$ and parameter $0 \leq \beta \leq 1$, we build a list of candidate vertices randomly chosen with size $\beta|V|$. This list is iterated once and the vertex is removed from G if and only if it is not an articulation point.

A total of a thousand instances were evaluated in our tests. In the analysis that follows, they are grouped by number of vertices and the group names use the format `[m:M]|A`, meaning that the group contains A instances where the number of vertices ranges from m to M". In total, 706 were tested and solved to optimality with the `EXACT` algorithm. The input and solution files of each instance in this benchmark can be downloaded at `http://www.ic.unicamp.br/~cid/Problem-instances/Milling`.

All tests have been executed in a machine with an Intel Core2 Quad Q9550 @ 2.83GHz processor and 8GB RAM. The `EXACT` algorithm was implemented with IBM ILOG CPLEX 12.1. In the current implementation CPLEX is also used in the separation routine due to its extensions to solve network flow problems. The `APX` algorithm uses the Blossom V [10] implementation of minimum cost perfect matching algorithm. All tests ran within a time limit of 1800s, and the number of iterations of `HEUR` was set to 2048, unless stated otherwise.

The main questions we tried to address with these experiments were (i) *How tight is the approximation factor of* `APX` *in practice?* (ii) *What is the behavior of the heuristic* `HEUR` *relative to the optimum?* and (iii) *How the heuristic compares to the approximation algorithm?* When comparing two algorithms A and B, we use the relative gap defined as $\frac{z_A - z_B}{z_B}$, where z_A, z_B are the objective values of algorithms A and B, respectively. In table 1 we compare algorithms `APX` and `HEUR` with `EXACT` for the set of instances with known optimum. One can notice how far is the `APX` relative gap from the theoretical value of 2.75 and the way that `HEUR` seems to be more sensible to variations in instance size when compared with `APX`. The standard deviation is quite high for the approximation algorithm while the heuristic approach seem to be more robust. Nevertheless, in both cases the maximum has deviated a lot from the average case.

Next we investigate how `HEUR` behaves as the maximum number of iterations changes. Table 2 shows the relative gap with relation to `EXACT` for the optimal instances, and this parameter is set to: 1024, 2048, 4096 and 8192. From these results, two observations are possible relative to the relation between the number

Table 1. Comparing HEUR and APX with EXACT for all instances whose the optimal value is known

Group	Rel. Gap APX	max	Rel. Gap HEUR	max
[13:16]|193	0.215 ± 0.225	0.750	0.000 ± 0.000	0.000
[21:29]|218	0.193 ± 0.149	0.667	0.004 ± 0.024	0.167
[30:39]|127	0.233 ± 0.136	0.571	0.016 ± 0.042	0.182
[40:49]|97	0.263 ± 0.138	0.700	0.043 ± 0.052	0.200
[50:59]|47	0.274 ± 0.137	0.667	0.102 ± 0.071	0.300
[60:69]|16	0.232 ± 0.163	0.500	0.119 ± 0.083	0.286
[70:76]|8	0.280 ± 0.092	0.455	0.160 ± 0.054	0.273

Table 2. Relative Gap of HEUR w.r.t EXACT for different values of iterations

Group	1024 iter	max	2048 iter	max	4096 iter	max	8192 iter	max
[13:16]|193	0.000 ± 0.000	0.000	0.000 ± 0.000	0.000	0.000 ± 0.000	0.000	0.000 ± 0.000	0.000
[21:29]|218	0.006 ± 0.029	0.167	0.005 ± 0.026	0.167	0.005 ± 0.026	0.167	0.004 ± 0.024	0.167
[30:39]|127	0.021 ± 0.047	0.182	0.014 ± 0.039	0.167	0.011 ± 0.034	0.167	0.009 ± 0.032	0.167
[40:49]|97	0.058 ± 0.058	0.250	0.041 ± 0.050	0.154	0.031 ± 0.046	0.154	0.022 ± 0.041	0.125
[50:59]|47	0.123 ± 0.068	0.250	0.102 ± 0.067	0.250	0.082 ± 0.060	0.250	0.068 ± 0.062	0.250
[50:59]|47	0.136 ± 0.075	0.300	0.131 ± 0.070	0.300	0.097 ± 0.050	0.167	0.079 ± 0.055	0.167
[60:69]|16	0.154 ± 0.070	0.273	0.134 ± 0.077	0.273	0.122 ± 0.074	0.273	0.110 ± 0.054	0.182

Table 3. Five hardest instances for APX

Instance	Num. of Vertices	Obj. APX	Gap. HEUR @ 16384 iter	Gap. HEUR @ 100000 iter
T0	96	54	0.08	0.08
T1	307	130	-0.21	-0.23
T2	353	142	-0.07	-0.04
T3	180	108	0.05	0.08
T4	478	262	-0.22	-0.20

of iterations executed by the algorithm and the size of an instance. First, one can see that the number of iterations should be increased as the number of vertices increases. Secondly, as expected, at a certain point, despite the increment on the number of iterations, the improvement observed in quality is negligible.

The importance of determining a good compromise value for the total number of iterations in HEUR is further justified when we focus on bigger instances. Table 3 shows detailed information about five instances not included in the previous tests. Their number of vertices are orders of magnitude bigger and no reasonable solution has been found using EXACT (which, of course, was unable to solve them to optimality). We can see that increasing the number of iterations by 10 (and, consequently, the computation time by the same amount) a minute gain is observed in the quality of the solution produced.

In terms of execution time we have that HEUR and APX have similar times, with the latter outperforming the former for small instances. Clearly, the execution time of APX is dominated by calculation of the minimum weight perfect matching between strips endpoints. On the other side HEUR depends linearly on the number

of iterations. Of course, EXACT has the biggest execution time of all three, and it is expected to yield results in reasonable time, only for instances having up to 70 vertices.

7 Conclusion and Future Directions

This paper proposes a branch-and-cut algorithm to solve the ODMP exactly. The algorithm is based on an integer programming model and was able to solve instances of moderate size. There is plenty of room for improvements in the algorithm since a polyhedral investigation of the formulation is yet to be done. On the heuristic side, the possibilities for new developments are even larger since our experimental results showed that the non-exact algorithms of sections 4 and 5 yield solutions with large duality gaps. In this particular, we emphasize that the Gardener's algorithm presented in section 5 has no local search phase. Thus, finding efficient neighborhoods may lead to solutions of much higher quality. Both issues mentioned above are currently being investigated.

References

1. Arkin, E.M., Bender, M.A., Demaine, E., Fekete, S.P., Mitchell, J.S.B., Sethia, S.: Optimal covering tours with turn costs. In: Proc. 13th ACM-SIAM Symposium on Discrete Algorithms, pp. 138–147 (2001)
2. Demaine, E.D., Mitchell, J.S.B., O'Rourke, J.: The open problems project. http://maven.smith.edu/~orourke/TOPP (page visited in January, 2011)
3. Arkin, E., Bender, M., Demaine, E., Fekete, S., Mitchell, J., Sethia, S.: Optimal covering tours with turn costs. SIAM Journal on Computing 35(3), 531–566 (2005)
4. Clossey, J., Laporte, G., Soriano, P.: Solving arc routing problems with turn penalties. Journal of the Operational Research Society 52, 433–439 (2001)
5. Benavent, E., Soler, D.: The directed rural postman problem with turn penalties. Transportation Science 30(4), 408–418 (1999)
6. Grötschel, M., Lovász, L., Schrijver, A.: The ellipsoid method and its consequences in combinatorial optimization. Combinatorica 1(2), 169–197 (1981)
7. Ben-Arieh, D., Gutin, G., Penn, M., Yeo, A., Zverovitch, A.: Transformations of generalized ATSP into ATSP. Operations Research Letters 31(5), 357–365 (2003)
8. Bondy, J.A., Murty, U.S.R.: Graph Theory with Applications. Elsevier Science, New York (1976)
9. Resende, M.G., Ribeiro, C.C.: Greedy randomized adaptive search procedures: Advances, hybridizations, and applications. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics. International Series in Operations Research and Management Science, vol. 146, pp. 283–319. Springer, US (2010)
10. Kolmogorov, V.: Blossom V: A new implementation of a minimum cost perfect matching algorithm. Mathematical Programming Computation 1(1), 43–67 (2009)

A Branch-Cut-and-Price Algorithm for the Capacitated Arc Routing Problem

Rafael Martinelli[1], Diego Pecin[1], Marcus Poggi[1], and Humberto Longo[2]

[1] PUC-Rio – Departamento de Informática
Rua Marquês de São Vicente, 225 RDC – Rio de Janeiro, RJ 22453-900, Brazil
{rmartinelli,dpecin,poggi}@inf.puc-rio.br
[2] UFG – Instituto de Informática
Campus Samambaia, Goiânia, GO 74001-970, Brazil
longo@inf.ufg.br

Abstract. Arc routing problems are among the most challenging combinatorial optimization problems. We tackle the Capacitated Arc Routing Problem where demands are spread over a subset of the edges of a given graph, called the required edge set. Costs for traversing edges, demands on the required ones and the capacity of the available identical vehicles at a vertex depot are given. Routes that collect all the demands at minimum cost are sought. In this work, we devise a Branch-Cut-and-Price algorithm for the Capacitated Arc Routing problem using a column generation which generates non-elementary routes (usually called *q-routes*) and exact separation of odd edge cutset and capacity cuts. Computational experiments report one new optimal and twelve new lower bounds.

Keywords: Arc Routing, Branch-Cut-and-Price, Integer Programming.

1 Introduction

The Capacitated Arc Routing Problem (CARP) is a problem which has several applications in the real life. One can think of this problem as a garbage collection problem. Suppose a company is responsible for collecting the garbage of a neighborhood. This company has a set of identical vehicles available in its base and knows a priori the amount of garbage generated on each street. As a company, its objective is to minimize the operational costs. So, the company needs to define the route for each vehicle which gives the lowest distance traveled. These routes must begin and end at the company's base, none of the vehicles can have its capacity violated, the garbage of a given street may be collected only by a single vehicle and all the garbage must be collected. Other possible applications for this problem are street sweeping, winter gritting, electric meter reading and airline scheduling [35].

This problem is strongly NP-Hard as shown by Golden and Wong in 1981 [16], where it was also proposed. Since then, several works were made with different approaches. We can cite some specific primal heuristics like Augment-Merge [16,15], Path-Scanning [15], Parallel-Insert [8], Construct-Strike [32] and Augment-Insert

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 315–326, 2011.
© Springer-Verlag Berlin Heidelberg 2011

[33]. There are several heuristics for obtaining lower bounds, like Matching [16], Node Scanning [3], Matching-Node Scanning [31], Node Duplication [19], Hierarchical Relaxations [2] and Multiple Cuts Node Duplication [36].

For the last years, most of the approximate results for the CARP were found using metaheuristics. All kinds of metaheuristics have already been proposed for the CARP, some of them are Simulated Annealing [12], the CARPET Tabu Search [18], Genetic Algorithm [22], Memetic Algorithm [23], Ant Colony [10], Guided Local Search [6] and Deterministic Tabu Search [7].

As being a hard problem, it is very difficult to solve the CARP to optimality. The works which tried to achieve this usually use integer programming. The first integer programming formulation was proposed by Golden and Wong [16] and since then some other formulations were proposed by Belenguer and Benavent [4] and Letchford [26], who also proposed valid inequalities for the problem. These integer formulations were solved using techniques such Branch-and-Bound [20], Cutting Plane [5,1], Column Generation [17,25] and Branch-and-Cut [24].

The objective of this work is to devise a Branch-Cut-and-Price algorithm for the CARP using a column generation algorithm with columns associated to non-elementary routes (also called *q-routes*) where 2-cycles are eliminated and exactly separating the odd edge cutset and capacity cuts. As far as we know, no Branch-Cut-and-Price algorithm which generates routes on the original graph was tailored for this problem. However, a Branch-Cut-and-Price was applied to the CARP by a transformation from the Capacitated Vehicle Routing Problem (CVRP) [29].

This work is divided in six sections. On section 2, we present mathematical formulations for the problem. On section 3, we detail the column generation with the non-elementary route pricing. On section 4, the Branch-Cut-and-Price is devised. On section 5, we show the computational experiments. On section 6, we conclude our work and point out future directions.

2 Mathematical Formulations

The CARP can be defined as follows. Consider a connected undirected graph $G = (V, E)$, with vertex set V and edge set E, costs $c : E \rightarrow \mathbb{Z}_0^+$, demands $d : E \rightarrow \mathbb{Z}_0^+$, a set I containing k identical vehicles with capacity Q and a distinguished depot vertex labeled 0. Define $E_R = \{e \in E | d_e > 0\}$ as the set of required edges. Let F be a set of closed routes which start and end at the depot, where the edges in a route can be either *serviced* or *deadheaded* (when the vehicle traverses the edge without servicing it). The set F is a feasible CARP solution if:

- Each required edge is serviced by exactly one route in F and
- The sum of demands of the serviced edges in each route F does not exceed the vehicle capacity.

We want to find a solution minimizing the sum of the costs of the routes. It corresponds to minimize the sum of the deadheaded edges' cost in the routes.

2.1 Two-Index Formulation

The most intuitive formulation for the CARP is to create a binary variable, x_e^k, for each required edge and each vehicle, an integer variable, z_e^k, for each deadheaded edge and each vehicle and use them in flow constraints. This formulation, known as two-index formulation [25], can be written as follows.

$$\text{MIN} \sum_{p \in I} \left(\sum_{e \in E_R} c_e x_e^p + \sum_{e \in E} c_e z_e^p \right) \tag{1}$$

$$\text{s.t.} \quad \sum_{p \in I} x_e^p = 1 \qquad \forall e \in E_R \tag{2}$$

$$\sum_{e \in E_R} d_e x_e^p \leq Q \qquad \forall p \in I \tag{3}$$

$$\sum_{e \in \delta_R(S)} x_e^p + \sum_{e \in \delta(S)} z_e^p \geq 2x_f^p \qquad \forall S \subseteq V \backslash \{0\}, f \in E_R(S), p \in I \tag{4}$$

$$\sum_{e \in \delta_R(S)} x_e^p + \sum_{e \in \delta(S)} z_e^p \equiv 0 \bmod (2) \quad \forall S \subseteq V \backslash \{0\}, p \in I \tag{5}$$

$$x_e^p \in \{0,1\} \qquad \forall e \in E_R, p \in I \tag{6}$$

$$z_e^p \in \mathbb{Z}_0^+ \qquad \forall e \in E, p \in I \; . \tag{7}$$

The objective function (1) minimizes the cost of each edge, being serviced or deadheaded. Constraints (2) assure that all required edges are serviced. Constraints (3) limit the total demand serviced by each vehicle to the capacity Q. Given S a vertex set, $E_R(S) = \{(i,j) \in E_R | i \in S, j \in S\}$, $\delta(S) = \{(i,j) \in E | i \in S, j \notin S\}$ and $\delta_R(S) = \{(i,j) \in E_R | i \in S, j \notin S\}$, constraints (4) assure that every route is connected and constraints (5) force every route in the solution to induce an Eulerian graph.

This formulation can be used to generate complete solutions for the CARP, but there is an exponential number of constraints (5). The separation of these constraints turns this formulation prohibitive when one is trying to solve large instances of the problem.

2.2 One-Index Formulation

A relaxation for the CARP is given by the one-index formulation, defined in [5]. This formulation considers only the deadheaded edges and aggregates every vehicle in just one variable. So, there is an integer variable z_e, which indicates the number of times an edge e is deadheaded by *any* vehicle. This formulation uses valid inequalities as constraints, since none of the two-index formulation's constraints can be used with these variables.

Valid inequalities (cuts) are added to formulations in order to improve the solution. Two families of cuts are widely used in almost any formulation for the CARP. The first one was created knowing that it is easy to show that every cutset S must have an even degree on any feasible solution. Because of that, for any cutset S containing an odd number of required edges, i.e. odd $|\delta_R(S)|$, at

least one edge $e \in \delta(S)$ will be deadheaded. These are called *Odd Edge Cutset constraints*:

$$\sum_{e \in \delta(S)} z_e \geq 1 \quad \forall S \subseteq V \backslash \{0\}, |\delta_R(S)| \text{ odd} . \tag{8}$$

The second family of cuts comes from the knowledge of a lower bound for the number of vehicles needed to service the required edges of a cutset, called $k(S)$. This lower bound can be obtained solving a Bin Packing for each cutset S, but since this problem is NP-Hard [14], it is better to obtain an approximation for this value. A very good approximation is obtained dividing the sum of demands of each required edge $e \in E_R(S) \cup \delta_R(S)$ by the vehicle capacity, as shown in equation (9).

$$k(S) = \left\lceil \frac{\sum_{e \in E_R(S) \cup \delta_R(S)} d_e}{Q} \right\rceil . \tag{9}$$

Knowing that at least $2k(S)$ vehicles must cross any cutset S, we can conclude that at least $2k(S) - |\delta_R(S)|$ edges from the cutset will be deadheaded in any feasible solution. These are the *Capacity constraints*:

$$\sum_{e \in \delta(S)} z_e \geq 2k(S) - |\delta_R(S)| \quad \forall S \subseteq V \backslash \{0\} . \tag{10}$$

As the left hand side of both (8) and (10) constraints are the same, we can put them together in just one constraint with the right hand side shown in (11).

$$\alpha(S) = \begin{cases} \max\{2k(S) - |\delta_R(S)|, 1\} \text{ if } |\delta_R(S)| \text{ is odd,} \\ \max\{2k(S) - |\delta_R(S)|, 0\} \text{ if } |\delta_R(S)| \text{ is even.} \end{cases} . \tag{11}$$

With everything defined, we can state the one-index formulation:

$$\text{MIN} \sum_{e \in E} c_e z_e \tag{12}$$

$$\text{s.t.} \sum_{e \in \delta(S)} z_e \geq \alpha(S) \; \forall S \subseteq V \backslash \{0\} \tag{13}$$

$$z_e \in \mathbb{Z}_0^+ \qquad \forall e \in E . \tag{14}$$

The objective function (12) minimizes the deadheadeds cost. In order to obtain the total cost, one must sum this value with the required edges costs. Constraints (13) are the odd edge cutset and capacity constraints together.

This formulation is a relaxation for the CARP because it does not give a complete solution for the problem and sometimes the solution found does not correspond to any feasible solution. But in practice, it gives very good lower bounds. The difficulty which arises here is how to generate the cuts (8) and (10). We will discuss this later.

2.3 Set Partitioning Approach

Another way to formulate the problem is, given a set of every possible route Ω, create a binary variable λ_r for every route $r \in \Omega$ and use them within a set partitioning formulation. Let the binary constant a_r^e be 1 if route r services the required edge e, 0 otherwise, and the integer constant b_r^e be the number of times edge e is deadheaded in route r. The set partitioning formulation for the CARP is as follows.

$$\text{MIN} \sum_{r \in \Omega} c_r \lambda_r \tag{15}$$

$$\text{s.t.} \quad \sum_{r \in \Omega} \lambda_r = k \tag{16}$$

$$\sum_{r \in \Omega} a_r^e \lambda_r = 1 \ \forall e \in E_R \tag{17}$$

$$\lambda_r \in \{0, 1\} \quad \forall r \in \Omega \ . \tag{18}$$

The objective function (15) minimizes the total cost of the used routes. Constraints (16) limit the number of routes used to the number of available vehicles and constraints (17) assure each required edge is serviced by only one route.

This formulation is obtained doing a Dantzig-Wolfe decomposition on constraints (3), (4) and (5) from the two-index formulation. The decomposition of these constraints defines the λ_r variables as routes. This is the reason why there is no vehicle index on the variables. Remark that this decomposition does not enforces the routes to be elementary.

Knowing how many times a route r traverses an edge e as deadheaded, we can create a direct mapping between the λ_r variables and the z_e variables from the one-index formulation, as shown in (19). This is enough to use the one-index constraints (13) to improve this formulation.

$$\sum_{r \in \Omega} b_r^e \lambda_r = z_e \quad \forall e \in E \ . \tag{19}$$

As the former two formulations, this one also has a problem. The number of possible routes is exponentially large, making it hard to generate all of them. To tackle this difficulty, we use a column generation algorithm.

3 Column Generation

Column generation is a technique for solving a linear program which has a prohibitive number of variables (columns). The algorithm starts with a small set of columns and solves the linear program (called here restricted master) to optimality. After that, the algorithm 'prices' the columns trying to find one with a reduced cost suitable to improve the solution. It then repeats the whole operation until no improving column is found.

In order to generate columns for the set partitioning formulation, first we have to define the reduced cost of a route. Given the dual variables γ, β_e and π_S, associated with constraints (16), (17) and (13), the reduced cost of a route r is:

$$\tilde{c}_r = c_r - \gamma - \sum_{e \in E_R} a_r^e \beta_e - \sum_{S \subseteq V \setminus \{0\}} \sum_{e \in \delta(S)} b_r^e \pi_S \tag{20}$$

$$= -\gamma + \sum_{e \in E_R} a_r^e (c_e - \beta_e) + \sum_{e \in E} b_r^e \left(c_e - \sum_{S \subseteq V \setminus \{0\} : e \in \delta(S)} \pi_S \right) . \tag{21}$$

The objective of the pricing subproblem is to find a route r (not necessarily elementary) with the smallest reduced cost $\tilde{c_r}$. This corresponds to solve a *Shortest Path Problem with Resource Constraints* (SPPRC), problem which can be solved in pseudopolynomial time with dynamic programming [9].

Our basic algorithm starts with a dynamic programming matrix $T(e, c, v)$, which indicates the reduced cost of the path from the depot to required edge e, with capacity c, ending at vertex v. It means that, for each edge e and capacity c, we have to store the value for both endpoints of e. Let $\tilde{c}_e = c_e - \beta_e$ and $\tilde{g}_e = c_e - \sum_{S \subseteq V \setminus \{0\} : e \in \delta(S)} \pi_S$ be the reduced costs associated with the required and deadheaded edges, respectively, $e = (w, v)$ and $f = (i, j)$. We initialize every cell with $+\infty$ and use the following recurrence to fill the dynamic programming matrix, which gives a complexity of $O(|E_R|^2 Q)$:

$$\begin{cases} T(e, c, v) = \min_{f \in E_R} \{T(f, c - d_e, i) + \tilde{c}_e + \tilde{g}_{iw}, T(f, c - d_e, j) + \tilde{c}_e + \tilde{g}_{jw}\} \\ T(e, d_e, v) = \tilde{c}_e + \tilde{g}_{0w} - \gamma \end{cases} . \tag{22}$$

In order to improve the lower bounds, we use a cycle elimination scheme, which forbids repeating edges within a given size. A 2-cycle elimination forbids cycles of size one $(e - e)$ and size two $(e - f - e)$. To do 1-cycle elimination, the algorithm avoids updating a required edge from the same edge. For 2-cycle elimination, it stores the two bests reduced costs from different edges in each cell of the dynamic programming matrix. To eliminate cycles greater than 2 is more complicated and was not used in this work. A complete description of k-cycle elimination can be found in [21].

Ideally, one would like to consider only elementary routes. With this modification the pricing subproblem turns into an *Elementary Shortest Path Problem with Resource Constraints* (ESPPRC), where the resource is the vehicle's capacity Q. This problem is strongly NP-Hard [11], but one can eventually solve it exactly when the graph is sparse [25].

4 Branch-Cut-and-Price

Since the column generation algorithm solves the linear relaxation of the set partitioning formulation, another technique is needed in order to obtain an integer

solution. In this work, we use Branch-and-Bound, a widely known technique, which enumerates all possible solutions through a search tree, discarding the branches with solutions greater than a known upper bound for the instance. This technique, when used together with column generation and cut separation is called *Branch-Cut-and-Price* (BCP).

The BCP algorithm starts on the root node of the search tree. The column generation algorithm is executed and a lower bound for the solution is found. If the solution is not integral, it applies a cut separation routine and re-execute the column generation algorithm. When no cuts are found, it branches. A variable with a continuous value is chosen and two branches are created, following a defined branching rule, and the nodes of these branches are put in a queue. There are some policies which can be used to choose the next node to explore. In our BCP algorithm, we put the nodes in a heap and always choose the one with the lowest value (lower bound). The whole procedure is then repeated for each node. The algorithm stops when the difference between the lower bound and the best integer solution found (upper bound) is less than one.

4.1 Branching Rule

The most intuitive branching rule is, given variable λ_r from the set partitioning formulation with a continuous solution $\bar{\lambda_r} \in [0,1]$, to create two branches, the first one with $\lambda_r = 0$ and other with $\lambda_r = 1$. But this cannot be done due to the column generation algorithm. If a route r is fixed to zero, the pricing subproblem will find this route again on the next iteration and will return it to the restricted master.

There are some ways to cope with this difficulty. But instead of that, we prefer to branch on the deadheaded edges variables. When we need to branch, we obtain the values of the z_e variables using equation (19), search for the variable whose value $\bar{z}_e$ is closer to 0.5 and then create two branches, one with $z_e \leq \lfloor \bar{z}_e \rfloor$ and other with $z_e \geq \lceil \bar{z}_e \rceil$. Mapping these to λ_r variables, we get:

$$\sum_{r \in \Omega} b_r^e \lambda_r \leq \lfloor \bar{z}_e \rfloor \tag{23}$$

$$\sum_{r \in \Omega} b_r^e \lambda_r \geq \lceil \bar{z}_e \rceil \ . \tag{24}$$

These inequalities generate new bounds constraints on the restricted master problem:

$$lb_e \leq \sum_{r \in \Omega} b_r^e \lambda_r \leq ub_e \quad \forall e \in E \ . \tag{25}$$

Note that when an integer solution is found, it is integer just on z_e variables and it may not be integer on λ_r. For this reason, this integer solution may be lower than the optimal solution and may also be infeasible – it behaves like the

one-index formulation. Nevertheless, these integer solutions give very good lower bounds for the problem [5].

4.2 Pricing

With the introduction of the bounds constraints, the pricing subproblem does not change, but the reduced cost of a route (21) must consider the dual values associated with these constraints. Given ρ_e, the dual variable associated with constraints (25), the new equation for the reduced cost of a route is:

$$\tilde{c}_r = -\gamma + \sum_{e \in E_R} a_r^e \left(c_e - \beta_e\right) + \sum_{e \in E} b_r^e \left(c_e - \rho_e - \sum_{S \subseteq V \setminus \{0\} : e \in \delta(S)} \pi_S\right) . \quad (26)$$

4.3 Strong Branching

In order to obtain better lower bounds faster, the BCP algorithm does strong branching. When a branching variable needs to be chosen, it selects n candidates to branch (here $n = 3$), usually the first n closest to 0.5. Then, it runs the column generation algorithm for both branches of every candidate and, given $left_c$ and $right_c$, the values of left and right branches of candidate c, it chooses the one with largest $min\{left_c, right_c\}$. In the case of a tie, it chooses the one with largest $max\{left_c, right_c\}$. If it finds a candidate with at least one branch infeasible, this one is chosen immediately.

4.4 Cut Generation

Our BCP algorithm pre-generates cuts before starting to solve the root node of the search tree. We use the one-index formulation and separate (8) and (10) cuts using the algorithms described in [30] and [1] respectively. The first one runs several maximum flows, which can be done in polynomial time. The second one is more difficult and is solved using mixed integer programming. This cut generation runs iteratively until no new cut is found. After that, we gather the cuts and start to solve the root node of the BCP.

During the BCP algorithm, for each node open on the search tree, we run the separation algorithm for the (8) cuts. Only when an integer solution is found, we run the separation algorithm for the (10) cuts. It would be prohibitively costly to run this separation on every node of the BCP.

5 Computational Experiments

All algorithms were implemented in C++, using Windows Vista Business 32-bits, Visual C++ 2008 Express Edition and IBM Cplex 12.2. Tests were conducted on an Intel Core 2 Duo 2.8 GHz, using just one core, with 4GB RAM. We applied

Table 1. Results for eglese instances

Name	$\|V\|$	$\|E_R\|$	$\|E\|$	k	UB	LB	Cut Generation			Branch-Cut-and-Price				
							Cost	Cuts	Time	Root	Cost	Cuts	Nodes	Time
e1-a	77	51	98	5	3548	3548	3527	55	44	3543	**3548**	3	2	151
e1-b	77	51	98	7	4498	4498	4464	78	51	4465	4487	9	70	436
e1-c	77	51	98	10	5595	5566	5513	70	52	5528	5537	0	49	158
e2-a	77	72	98	7	5018	5018	4995	79	34	5002	5012	73	49	1167
e2-b	77	72	98	10	6317	6305	6271	78	38	6280	6291	31	143	996
e2-c	77	72	98	14	8335	8243	8161	80	49	8228	**8274**	58	4179	21600
e3-a	77	87	98	8	5898	5898	5894	73	31	5895	**5898**	1	2	746
e3-b	77	87	98	12	7775	7704	7649	75	45	7687	**7715**	116	1755	18986
e3-c	77	87	98	17	10292	10163	10125	84	51	10176	**10207**	91	3505	21600
e4-a	77	98	98	9	6444	6408	6378	71	22	6389	6395	120	63	3025
e4-b	77	98	98	14	8962	8884	8838	70	37	8874	**8893**	223	1947	21600
e4-c	77	98	98	19	11550	11427	11376	76	50	11439	**11471**	133	2719	21600
s1-a	140	75	190	7	5018	5018	5010	143	331	5013	5014	6	3	318
s1-b	140	75	190	10	6388	6384	6368	154	451	6376	**6388**	62	64	3557
s1-c	140	75	190	14	8518	8493	8404	153	806	8457	**8494**	134	1157	21600
s2-a	140	147	190	14	9884	9824	9737	131	449	9796	9807	537	147	21600
s2-b	140	147	190	20	13100	12968	12901	145	718	12950	**12970**	333	387	21600
s2-c	140	147	190	27	16425	16353	16248	154	1683	16331	**16357**	327	743	21600
s3-a	140	159	190	15	10220	10143	10083	129	303	10131	**10146**	344	157	21600
s3-b	140	159	190	22	13682	13616	13568	127	574	13608	**13623**	372	319	21600
s3-c	140	159	190	29	17194	17100	17007	135	1931	17088	**17115**	420	615	21600
s4-a	140	190	190	19	12268	12143	12026	121	343	12121	12140	270	159	21600
s4-b	140	190	190	27	16283	16093	15984	135	604	16049	16082	498	294	21600
s4-c	140	190	190	35	20517	20375	20236	140	1165	20362	**20380**	252	581	21600
mean					9738.7	9673.8	9615.1	106.5	410.9	9657.8	9676.8	183.9	795.2	13830.8
opt					—	5	0	—	—	0	3	—	—	—
gap					—	0.564%	1.152%	—	—	0.767%	0.561%	—	—	—

our algorithms to the instances of the dataset *eglese*, which was originally used in [27] and [28]. These instances were constructed using as underlying graph regions of the road network of the county of Lancashire (UK). They used cost and demands proportional to the length of the edges and most of the instances have non-required edges. From the classic sets of instances (*kshs, gdb, val* and *eglese*), this one is the only one which still has open instances.

Results are shown in table 1. Column *UB* lists the best known upper bounds, reported by Santos et al. [34] and Fu et al. [13]. Column *LB* lists the best known lower bounds, reported by Longo et al. [29] and Brandão and Eglese [7]. This latter work is a primal work and reported some lower bounds which we could not find in any paper, even in the ones referred by them. These lower bounds have been widely used, with rare exceptions, for instance in Santos et al. [34]. The next 3 columns show results for the cut generation done before solving the root node. Columns *Cost*, *Cuts* and *Time* show the solution cost, the number of cuts sent to BCP and the solution time in seconds. The next 5 columns show results for the BCP algorithm. Columns *Root*, *Cost*, *Cuts*, *Nodes* and *Time* show the solution cost at the root node, the BCP solution cost, the number of cuts generated, the nodes opened and the total time in seconds.

In order to run our BCP algorithm for all instances, we set the time limit of 6 hours (21600 seconds). At this point, the algorithm is stopped and the solution reported is the best found so far. All continuous values, costs or times, were rounded up to the next integer.

The improving lower bounds found are shown in bold font and optimal values are underlined. The BCP algorithm found 3 optimal values, *e1-a* and *e3-a* were already known and *s1-b* is a new optimal value.

6 Conclusions and Future Research

We have created the first Branch-Cut-and-Price algorithm for the Capacitated Arc Routing Problem which generates routes on the original graph. The algorithm could find one optimal value and twelve new lower bounds for the eglese instances in reasonable time.

As future research, there are some techniques that may be further explored. We can cite some examples. A column generation algorithm which generates only elementary routes can be tested with the Branch-Cut-and-Price. An enumeration routine which generates all routes between the lower and upper bound can be used to find the optimum on some instances. Besides that, some techniques may be used to speed up the Branch-Cut-and-Price algorithm, like active reduced cost fixing.

Acknowledgements. The contribution by Rafael Martinelli, Diego Pecin and Marcus Poggi de Aragão has been partially supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), processes numbers 140849 / 2008-4, 141538 / 2010-4 and 309337 / 2009-7.

References

1. Ahr, D.: Contributions to Multiple Postmen Problems. Ph.D. dissertation, Department of Computer Science, Heidelberg University (2004)
2. Amberg, A., Voß, S.: A Hierarchical Relaxations Lower Bound for the Capacitated Arc Routing Problem. In: Proceedings of the 35th Anual Hawaii International Conference on System Sciences (2002)
3. Assad, A.A., Pearn, W.L., Golden, B.L.: The Capacitated Chinese Postman Problem: Lower Bounds and Solvable Cases. American Journal of Mathematical Management Sciences 7(1,2), 63–88 (1987)
4. Belenguer, J.M., Benavent, E.: Polyhedral Results on the Capacitated Arc Routing Problem, Departamento de Estadística e Investigación Operativa, Universitat de València, Tech. Rep. TR 1-92 (1992)
5. Belenguer, J.M., Benavent, E.: A Cutting Plane Algorithm for the Capacitated Arc Routing Problem. Computers & Operations Research 30, 705–728 (2003)
6. Beullens, P., Muyldermans, L., Cattrysse, D., Oudheusden, D.V.: A Guided Local Search Heuristic for the Capacitated Arc Routing Problem. European Journal of Operational Research 147(3), 629–643 (2003)
7. Brandão, J., Eglese, R.: A Deterministic Tabu Search Algorithm for the Capacitated Arc Routing Problem. Computers & Operations Research 35, 1112–1126 (2008)
8. Chapleau, L., Ferland, J.A., Lapalme, G., Rousseau, J.M.: A Parallel Insert Method for the Capacitated Arc Routing Problem. Operations Research Letters 3(2), 95–99 (1984)
9. 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)
10. Doerner, K., Hartl, R., Maniezzo, V., Reimann, M.: An Ant System Metaheuristic for the Capacitated Arc Routing Problem. In: Proceedings of the 5th Metaheuristics International Conference, Tokyo, Japan (2003)
11. Dror, M.: Note on the Complexity of the Shortest Path Models for Column Generation in VRPTW. Operations Research 42(5), 977–978 (1994)
12. Eglese, R.W.: Routeing Winter Gritting Vehicles. Discrete Applied Mathematics 48(3), 231–244 (1994)
13. Fu, H., Mei, Y., Tang, K., Zhu, Y.: Memetic algorithm with heuristic candidate list strategy for capacitated arc routing problem. In: IEEE Congress on Evolutionary Computation, pp. 1–8 (2010)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
15. Golden, B.L., DeArmon, J.S., Baker, E.K.: Computational Experiments with Algorithms for a Class of Routing Problems. Computers and Operations Research 10(1), 47–59 (1983)
16. Golden, B.L., Wong, R.T.: Capacitated Arc Routing Problems. Networks 11, 305–315 (1981)
17. Gómez-Cabrero, D., Belenguer, J.M., Benavent, E.: Cutting Plane and Column Generation for the Capacitated Arc Routing Problem. In: ORP3, Valencia (2005)
18. Hertz, A., Laporte, G., Mittaz, M.: A Tabu Search Heuristic for the Capacitated Arc Routing Problem. Operations Research 48(1), 129–135 (2000)
19. Hirabayashi, R., Nishida, N., Saruwatari, Y.: Node Dublication Lower Bounds for the Capacitated Arc Routing Problems. Journal of the Operations Research Society of Japan 35(2), 119–133 (1992)

20. Hirabayashi, R., Nishida, N., Saruwatari, Y.: Tour Construction Algorithm for the Capacitated Arc Routing Problem. AsiaPacific Journal of Operational Research 9, 155–175 (1992)
21. Irnich, S., Villeneuve, D.: The Shortest-Path Problem with Resource Constraints and k-Cycle Elimination for $k \geq 3$. INFORMS Journal on Computing 18(3), 391–406 (2006)
22. Lacomme, P., Prins, C., Ramdane-Chérif, W.: A genetic algorithm for the capacitated arc routing problem and its extensions. In: Boers, E.J.W., Gottlieb, J., Lanzi, P.L., Smith, R.E., Cagnoni, S., Hart, E., Raidl, G.R., Tijink, H. (eds.) EvoIASP 2001, EvoWorkshops 2001, EvoFlight 2001, EvoSTIM 2001, EvoCOP 2001, and EvoLearn 2001. LNCS, vol. 2037, pp. 473–483. Springer, Heidelberg (2001)
23. Lacomme, P., Prins, C., Ramdane-Chérif, W.: Competitive memetic algorithms for arc routing problems. Annals of Operations Research 131, 159–185 (2004)
24. Laganá, D., Ghiani, G., Musmanno, R., Laporte, G.: A Branch-and-Cut Algorithm for the Undirected Capacitated Arc Routing Problem. The Journal of the Operations Research Society of America, 1–21 (2007)
25. Letchford, A., Oukil, A.: Exploiting Sparsity in Pricing Routines for the Capacitated Arc Routing Problem. Computers & Operations Research 36, 2320–2327 (2009)
26. Letchford, A.N.: Polyhedral Results for some Constrained Arc-Routing Problems, Ph.D. dissertation, Lancaster University (1996)
27. Li, L.Y.O.: Vehicle Routeing for Winter Gritting, Ph.D. dissertation, Department of Management Science, Lancaster University (1992)
28. Li, L.Y.O., Eglese, R.W.: An Interactive Algorithm for Vehicle Routeing for Winter-Gritting. Journal of the Operational Research Society 47, 217–228 (1996)
29. Longo, H., Poggi de Aragão, M., Uchoa, E.: Solving Capacitated Arc Routing Problems Using a Transformation to the CVRP. Computers & Operations Research 33, 1823–1827 (2006)
30. Padberg, M.W., Rao, M.R.: Odd minimum cut-sets and b-matchings. Mathematics of Operations Research 7, 67–80 (1982)
31. Pearn, W.L.: New Lower Bounds for the Capacitated Arc Routing Problem. Networks 18, 181–191 (1988)
32. Pearn, W.L.: Approximate Solutions for the Capacitated Arc Routing Problem. Computers & Operations Research 16(6), 589–600 (1989)
33. Pearn, W.L.: Augment-Insert Algorithms for the Capacitated Arc Routing Problem. Computers & Operations Research 18(2), 189–198 (1991)
34. Santos, L., Coutinho-Rodrigues, J., Current, J.: An Improved Ant Colony Optimization Based Algorithm for the Capacitated Arc Routing Problem. Transportation Research Part B 44, 246–266 (2010)
35. Wøhlk, S.: Contributions to Arc Routing, Ph.D. dissertation, Faculty of Social Sciences, University of Southern Denmark (2005)
36. Wøhlk, S.: New Lower Bound for the Capacitated Arc Routing Problem. Computers & Operations Research 33(12), 3458–3472 (2006)

A Biased Random Key Genetic Algorithm Approach for Unit Commitment Problem

Luís A.C. Roque[1], Dalila B.M.M. Fontes[2], and Fernando A.C.C. Fontes[3]

[1] ISEP-DEMA/GECAD, Instituto Superior de Engenharia do Porto, Portugal
[2] FEP/LIAAD-INESC Porto L.A., Universidade do Porto, Portugal
[3] FEUP/ISR-Porto, Universidade do Porto, Portugal
lar@isep.ipp.pt, fontes@fep.up.pt, faf@fe.up.pt

Abstract. A Biased Random Key Genetic Algorithm (BRKGA) is proposed to find solutions for the unit commitment problem. In this problem, one wishes to schedule energy production on a given set of thermal generation units in order to meet energy demands at minimum cost, while satisfying a set of technological and spinning reserve constraints. In the BRKGA, solutions are encoded by using random keys, which are represented as vectors of real numbers in the interval $[0,1]$. The GA proposed is a variant of the random key genetic algorithm, since bias is introduced in the parent selection procedure, as well as in the crossover strategy. Tests have been performed on benchmark large-scale power systems of up to 100 units for a 24 hours period. The results obtained have shown the proposed methodology to be an effective and efficient tool for finding solutions to large-scale unit commitment problems. Furthermore, from the comparisons made it can be concluded that the results produced improve upon some of the best known solutions.

Keywords: Unit Commitment, Genetic Algorithm, Optimization, Electrical Power Generation.

1 Introduction

The Unit Commitment (UC) is a complex optimization problem well known in the power industry and adequate solutions for it have potential large economic benefits that could result from the improvement in unit scheduling. Therefore, the UC problem plays a key role in planning and operating power systems. The thermal UC problem involves scheduling the turn-on and turn-off of the thermal generating units, as well as the dispatch for each on-line unit that minimizes the operating cost for a specific time generation horizon. In addition, there are multiple technological constraints, as well as system demand and spinning reserve constraints that must be satisfied. Due to its combinatorial nature, multi-period characteristics, and nonlinearities, this problem is highly computational demanding and, thus, it is a hard optimization task solving the UC problem, specially for real-sized systems.

The methodology proposed to find solutions for this problem is a Genetic Algorithm where the solutions are encoded using random keys. A Biased Random Key Genetic Algorithm (BRKGA) is proposed to find the on/off state of the generating units for every time period as well as the amount of production of each unit at each time period.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 327–339, 2011.
© Springer-Verlag Berlin Heidelberg 2011

In the past, several traditional heuristic approaches based on exact methods have been used such as dynamic programming, mixed-integer programming and benders decomposition, see e.g. [14,6,15]. However, more recently most of the developed methods are metaheuristics, evolutionary algorithms, and hybrids of the them, see e.g. [2,25,11,7,22,4,26,1]. These latter types have, in general lead to better results than the ones obtained with the traditional heuristics. The most used metaheuristic methods are simulated annealing (SA) [19,22], evolutionary programming (EP) [12,18], memetic algorithm (MA)[25], particle swarm optimization (PSO) [23,28] and genetic algorithms (GA), see e.g. [13,24]. Comprehensive and detailed surveys can be found in [16,20,17].

In this paper we focus on applying Genetic Algorithms (GAs) to find good quality solutions for the UC problem. The majority of the reported GA implementations to address the UC problem are based on the binary encoding. However, studies have shown that other encoding schemes such as real valued random keys [3] can be efficient when accompanied with suitable GA operators, specially for problems where the relative order of tasks is important. In the proposed algorithm a solution is encoded as a vector of N real random keys in the interval $[0,1]$, where N is the number of generation units. The Biased Random Key Genetic Algorithm (BRKGA) proposed in this paper is based on the framework provided by Resende and Gonçalves in [9]. BRKGAs are a variation of the Random key Genetic Algorithms (RKGAs), first introduced by Bean [3]. The bias is introduced at two different stages of the GA. On the one hand, when parents are selected we get a higher change of good solutions being chosen, since one of the parents is always taken from a subset including the best solutions. On the other hand, the crossover strategy is more likely to choose alleles from the best parent to be inherited by offspring. The work [9] provides a tutorial on the implementation and use of biased random key genetic algorithms for solving combinatorial optimization problems and many successful applications are reported in the references therein.

This paper is organized as follows. In Section 2, the UC problem is described and formulated, while in Section 3 the solution methodology proposed is explained. Section 4 describes the set of benchmark systems used in the computational experiments and reports on the results obtained. Finally, in Section 5 some conclusions are drawn.

2 UC Problem Formulation

In the UC problem one needs to determine at each time period the turn-on and turn-off times of the power generation units, as well as the generation output subject to operational constraints, while satisfying load demands at minimum cost. Therefore, we have two types of decision variables. The binary variables, which indicate the status of each unit at each time period and the real variables, which provide the information on the amount of energy produced by each unit at each time period. The choices made must satisfy two sets of constraints: the demand constraints (regarding the load requirements and the spinning reserve requirements) and the technical constraints (regarding generation unit constraints). The costs are made up two components: the fuel costs, i.e. production costs, and the start-up costs.

Let us now introduce the parameters and variables notation.

Decision Variables:
$\mathbf{Y_{t,j}}$: Thermal generation of unit j at time period t, in $[MW]$;
$\mathbf{u_{t,j}}$: Status of unit j at time period t (1 if the unit is on; 0 otherwise);
Auxiliary Variables:
$\mathbf{T_j^{on/off}(t)}$: Time periods for which unit j has been continuously on-line/off-line until time period t, in $[hours]$;
Parameters:
$\mathbf{T}$: Number of time periods (hours) of the scheduling time horizon;
$\mathbf{t}$: Time period index;
$\mathbf{N}$: Number of generation units;
$\mathbf{j}$: Generation unit index;
$\mathbf{R_t}$: System spinning reserve requirements at time period t, in $[MW]$;
$\mathbf{D_t}$: Load demand at time period t, in $[MW]$;
$\mathbf{Ymin_j}$: Minimum generation limit of unit j, in $[MW]$;
$\mathbf{Ymax_j}$: Maximum generation limit of unit j, in $[MW]$;
$\mathbf{N_b}$: Number of the base units;
$\mathbf{T_{min,j}^{on/off}}$: Minimum uptime/downtime of unit j, in $[hours]$;
$\mathbf{T_{c,j}}$: Cold start time of unit j, in $[hours]$;
$\mathbf{S_{H/C,j}}$: Hot/Cold start-up cost of unit j, in [\$];
$\Delta_j^{dn/up}$: Maximum allowed output level decrease/increase in consecutive periods for unit j, in $[MW]$;

2.1 Objective Function

As already said, there are two cost components: generation costs and start-up costs. The generation costs, i.e. the fuel costs, are conventionally given by a quadratic cost function as in equation (1), while the start-up costs, that depend on the number of time periods during which the unit has been off, are given as in equation (2).

$$F_j(Y_{t,j}) = a_j \cdot (Y_{t,j})^2 + b_j \cdot Y_{t,j} + c_j, \tag{1}$$

where a_j, b_j, c_j are the cost coefficients of unit j.

$$S_{t,j} = \begin{cases} S_{H,j} & \texttt{if } T_{min,j}^{off} \leq T_j^{off}(t) \leq T_{min,j}^{off} + T_{c,j} \\ S_{C,j} & \texttt{if } T_j^{off}(t) > T_{min,j}^{off} + T_{c,j} \end{cases}, \tag{2}$$

where $S_{H,j}$ and $S_{C,j}$ are the hot and cold start-up costs of unit j, respectively.

Therefore, the cost incurred with an optimal scheduling is given by the minimization of the total costs for the whole planning period, as in equation (3).

$$\text{Minimize} \quad \sum_{t=1}^{T} \left(\sum_{j=1}^{N} \left\{ F_j(Y_{t,j}) \cdot u_{t,j} + S_{t,j} \cdot (1 - u_{t-1,j}) \cdot u_{t,j} \right\} \right). \tag{3}$$

2.2 Constraints

The constraints can be divided into two sets: the demand constraints and the technical constraints. Regarding the first set of constraints it can be further divided into load requirements and spinning reserve requirements, which can be written as follows:

1) Power Balance Constraints
The total power generated must meet the load demand, for each time period.

$$\sum_{j=1}^{N} Y_{t,j} \cdot u_{t,j} \geq D_t, t \in \{1, 2, ..., T\}. \tag{4}$$

2) Spinning Reserve Constraints
The spinning reserve is the total amount of real power generation available from on-line units net of their current production level.

$$\sum_{j=1}^{N} Ymax_j \cdot u_{t,j} \geq R_t + D_t, t \in \{1,2,...,T\}. \quad (5)$$

The second set of constrains includes unit output range, minimum number of time periods that the unit must be in each status (on-line and off-line), and the maximum output variation allowed for each unit.

3) Unit Output Range Constraints
Each unit has a maximum and minimum production capacity.

$$Ymin_j \cdot u_{t,j} \leq Y_{t,j} \leq Ymax_j \cdot u_{t,j}, \text{for } t \in \{1,2,...,T\} \text{ and } j \in \{1,2,...,N\}. \quad (6)$$

4) Ramp rate Constraints
Due to the thermal stress limitations and mechanical characteristics the output variation, levels of each online unit in two consecutive periods are restricted by ramp rate limits.

$$-\Delta_j^{dn} \leq Y_{t,j} - Y_{t-1,j} \leq \Delta_j^{up}, \text{for } t \in \{1,2,...,T\} \text{ and } j \in \{1,2,...,N\}. \quad (7)$$

5) Minimum Uptime/Downtime Constraints
The unit cannot be turned on or off instantaneously once it is committed or uncommitted. The minimum uptime/downtime constraints indicate that there will be a minimum time before it is shut-down or started-up, respectively.

$$T_j^{on}(t) \geq T_{min,j}^{on} \text{ and } T_j^{off}(t) \geq T_{min,j}^{off}, \text{for } t \in \{1,2,...,T\} \text{ and } j \in \{1,2,...,N\}. \quad (8)$$

3 Methodology

Genetic Algorithms are a global optimization technique based on natural genetics and evolution mechanisms such as survival of the fittest law, genetic recombination and selection [10,8]. GAs provide great modeling flexibility and can easily be implemented to search for solutions of combinatorial optimization problems. Several GAs have been proposed for the unit commitment problem, see e.g. [13,5,24,2,27,7,1], the main differences being the representation scheme, the decoding procedure, and the solution evaluation procedure (i.e. fitness function).

Many GA operators have been used; the most common being copy, crossover, and mutation. Copy consists of simply copying the best solutions from the previous generation into the next one, with the intention of preserving the chromosomes corresponding to best solutions in the population. Crossover produces one or more *offspring* by combining the genes of solutions chosen to act as their parents. The mutation operator randomly changes one or more genes of a given chromosome in order to introduce some extra variability into the population and thus, prevent premature convergence.

The GA proposed here, i.e. the BRKGA, uses the framework proposed by Gonçalves and Resende in [9]. The algorithm evolves a population of chromosomes that are used

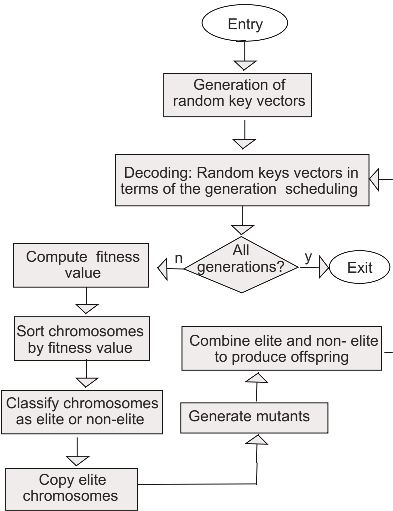


Fig. 1. The BRKGA framework

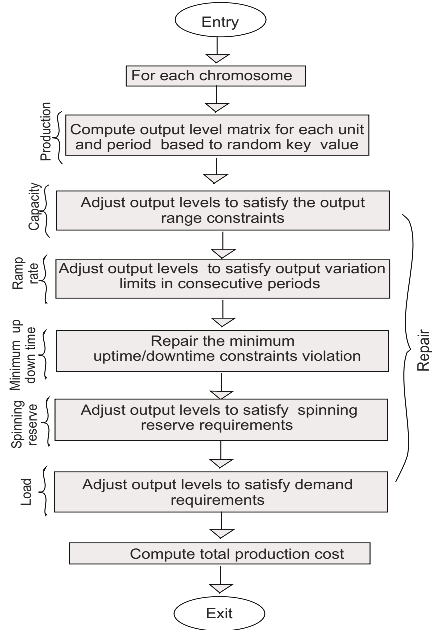


Fig. 2. Flow chart of the decoder

to assign priorities to the generation units. These chromosomes are vectors, of size N (number of units), of real numbers in the interval $[0,1]$ (called random keys). A new population is obtained by joining three subsets of solutions as follows: the first subset is obtained by copying the best solutions of the current population; the second subset is obtained by using a (biased) parameterized uniform crossover; the remaining solutions, termed mutants, are randomly generated as was the case for the initial population. The BRKGA framework is illustrated in Figure 1, which has been adapted from [9].

Specific to our problem are the decoding procedure, as well as the fitness computation. The decoding procedure, that is how solutions are constructed once a population of chromosomes is given, is performed in two main steps, as it can be seen in Figure 2. Firstly, a solution satisfying the load demand, for each period is obtained. In this solution, the units production is proportional to their priority, which is given by the random key value. Then, these solutions are checked for constraints satisfaction.

3.1 Decoding Procedure

Given a vector of numbers in the interval $[0,1]$, say $RK = (r_1, r_2, \ldots, r_N)$, percent vectors $V' = \left(v'_1, v'_2, \ldots, v'_{N_b}\right)$, $V = (v_1, v_2, \ldots, v_N)$ and rank vector $O = (o_1, o_2, \ldots, o_N)$ are computed. Each element v_j is computed as $v_j = \frac{r_j}{\sum_i^N r_i}, j = 1, 2, \ldots, N$ and v'_j is given by $v'_j = \frac{r_j}{\sum_i^{N_b} r_i}, j = 1, 2, \ldots, N_b$, where N_b is the number of base units, while each o_i is defined taking into account the descending order of the RK value.

Then an output generation matrix Y is obtained, where each element $Y_{t,j}$ gives the production level of unit $j = o_i, i = 1,...,N$ for time period t and is computed as the product of the percentage vectors V' or V by the periods demand D_t, as illustrated in the following pseudocode:

Algorithm 1. Initial matrix generation output

```
i = 1
d = D_t
while i ≤ N and d > 0 do
  j = o_i
  if j ≤ N_b then
    if D_t + R_t ≤ Σ_{k=1}^{N_b} Ymax_k then
      Y_{t,j} = v'_j.D_t
      d = d − Y_{t,j}
    else {D_t + R_t > Σ_{k=1}^{N_b} Ymax_k}
      Y_{t,j} = Ymax_j
      d = d − Y_{t,j}
    end if
  else {j > N_b }
    if D_t + R_t ≤ Σ_{k=1}^{N_b} Ymax_k then
      Y_{t,j} = 0
    else {D_t + R_t > Σ_{k=1}^{N_b} Ymax_k}
      if d > Ymax_j then
        Y_{t,j} = v_j.D_t
        d = d − Y_{t,j}
      else {d < Ymax_j}
        Y_{t,j} = d
        d = 0
      end if
    end if
  end if
  Next i
end while
```

The production level of unit j for each time period t however, may not be admissible and therefore, the solution obtained may be unfeasible. Hence, the decoding procedure also incorporates a repair mechanism. This mechanism forces constraints satisfaction.

The repair mechanism starts by forcing the output level of each unit to be in its output range, as given in equation (9).

$$Y_{t,j} = \begin{cases} Ymax_j & \text{if } Y_{t,j} > Ymax_j \\ Y_{t,j} & \text{if } Ymin_j \le Y_{t,j} \le Ymax_j \\ Ymin_j & \text{if } \chi \cdot Ymin_j \le Y_{t,j} < Ymin_j \\ 0 & \text{otherwise,} \end{cases} \tag{9}$$

where $\chi \in [0,1]$ is a scaling factor.

At the same time that the ramp constrains are ensured for a specific time period t, new output limits ($Y_{t,j}^{max}$ and $Y_{t,j}^{min}$ upper and lower limits, respectively) must be imposed, for the following period $t+1$, since their value depends on the output level of the current period t. Equations (10) and (11) show how this is done.

$$Y_{t,j} = \begin{cases} Y_{t,j}^{max} & \text{if } Y_{t,j} \geq Y_{t,j}^{max} \\ Y_{t,j} & \text{if } Y_{t,j}^{min} < Y_{t,j} < Y_{t,j}^{max} \\ Y_{t,j}^{min} & \text{if } \mu \cdot Y_{t,j}^{min} \leq Y_{t,j} \leq Y_{t,j}^{min} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

where $Y_{1,j}^{max} = Ymax_j$, $Y_{1,j}^{min} = Ymin_j$ and

$$Y_{t,j}^{max} = min\left\{Ymax_j, Y_{t-1,j} + \Delta_j^{up}\right\}, Y_{t,j}^{min} = max\left\{Ymin_j, Y_{t-1,j} - \Delta_j^{dn}\right\}. \tag{11}$$

After ensuring the unit output range constraints and the ramp rate constraints, it is still needed to guarantee that minimum up/down time constraints are satisfied. The adjustment of the unit status can be obtained using the repair mechanism illustrated in Figure 3. As it can be seen, for two consecutive periods the unit status can only be changed if the $T_{min}^{on/off}$ is already satisfied, for a previously turned on or off unit, respectively.

For each period, it may happen that the spinning reserve requirement is not satisfied. If the number of on-line units is not enough, some off-line units will be turned on, one at the time, until the cumulative capacity matches or is larger than $D_t + R_t$ as shown in

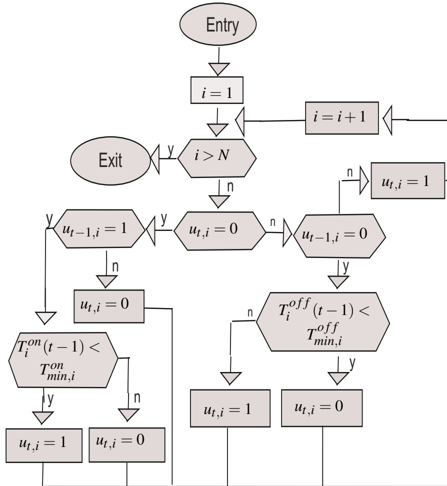


Fig. 3. Minimum up down time repair mechanism

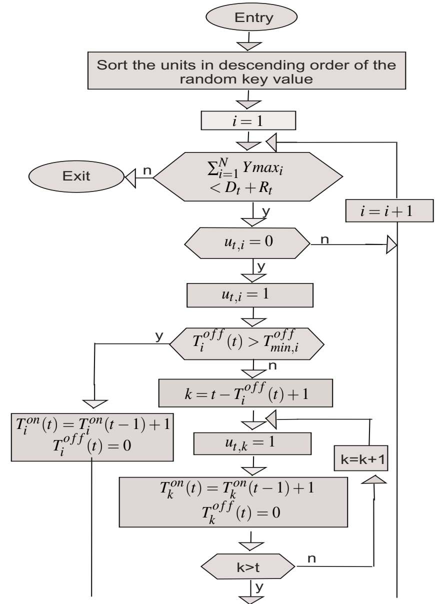


Fig. 4. Handling spinning reserve constraint

Figure 4. In doing so, units are considered in descending order of priority, i.e. random key value. After ensuring the spinning reserve satisfaction, it may happen that we end up with excessive spinning reserve. Since this is not desirable due to additional operational costs involved, we tried to decommit some units. Then units are considered for turning off-line, in ascending order of priority until their cumulative capacity reaches, $D_t + R_t$.

At the end of this procedure we have found the u and Y matrices specifying which units are being operated at each time period and how much its one is producing. However, it may happen that the production matches, is larger than, or lesser than the demand. In order to compute the total cost of the current solution we first must find how much each unit is producing.

If there is excessive production, the on-line units production is decreased to its minimum allowed value, one at the time, until either all are set to the minimum production or the production reaches the load demand value. In doing so, units are considered in descending order of priority, i.e. random key value. It should be notice that by reducing production at time period t the production limits at time period $t+1$ change, and the new values must be respected. Therefore, the minimum allowed production is given by $max\left\{Y_{t,j}^{min}, Y_{t+1,j} - \Delta_j^{up}\right\}$. This is repeated no more than N times. If there is lack of production, the on-line units production is increased to its maximum allowed value, one at the time, until either all are set to the maximum production or the production reaches the load demand value. In doing so, units are considered in ascending order of priority, i.e. random key value. It should be noticed that by increasing production at time period t the production limits at time period $t+1$ change, and the new values must be respected. Therefore, the maximum allowed production is given by $min\left\{Y_{t,j}^{max}, Y_{t+1,j} + \Delta_j^{dn}\right\}$. Again, this is repeated no more than N times. Once these repairing stages have been performed, the solutions obtained are feasible and the respective total cost is computed.

3.2 GA Configuration

The current population of solutions is evolved by the GA operators onto a new population as follows:

- 20% of the best solutions (elite set) of the current population are copied;
- 20% of the new population is obtained by introducing mutants, that is by randomly generating new sequences of random keys, which are then decoded to obtain mutant solutions. Since they are generated using the same distribution as the original population, no genetic material of the current population is brought in;
- Finally, the remaining 60% of the population is obtained by biased reproduction, which is accomplished by having both a biased selection and a biased crossover.

The selection is biased since, one of the parents is randomly selected from the elite set of solutions (of the current population), while the other is randomly selected from the remainder solutions. This way, elite solutions are given a higher chance of mating, and therefore of passing on their characteristics to future populations. Genes are chosen by using a biased uniform crossover, that is, for each gene a biased coin is tossed to decide on which parent the gene is taken from. This way, the offspring inherits the genes from the elite parent with higher probability (0.7 in our case).

4 Numerical Results

A set of benchmark systems has been used for the evaluation of the proposed algorithm. Each of the problems in the set considers a scheduling period of 24 hours. The set of systems comprises six systems with 10 up to 100 units. A base case with 10 units was initially defined, and the others have been obtained by considering copies of these units. The base 10 units system and corresponding 24 hours load demand are given in [13]. To generate the 20 units problem, the 10 original units have been duplicated and the load demand doubled. An analogous procedure was used to obtain the problems with 40, 60, 80, and 100 units. In all cases, the spinning reserve requirements were set to 10% of the load demand.

Several computational experiments were made in order to choose the parameter values. The BRKGA was implemented with biased crossover probability as main control parameter. The parameter ranges used in our experiments were $0.5 \leq P_c \leq 0.8$ with step size 0.1 which gives 4 possible values for biased crossover probability. The results obtained have shown no major differences. Nevertheless, the results reported here refer to the best obtained ones, for which the number of generations was set to $10N$, the population size was set to $min\{2N, 60\}$, the biased crossover probability was set to 0.7, and the scaling factor $\chi = 0.4$. Due to the stochastic nature of the BRKGA, each problem was solved 20 times. We compare the results obtained with the best results reported in literature. In tables 1, 2, and 3, we compare the best, average, and worst results obtained, for each of the six problems, with the best of each available in literature. As it can be seen, for two out of the six problems solved our best results improve upon the best known results, while for the other four it is within 0.02% and 0.18% of the best known solutions. Moreover when our algorithm is not the best, it is the second best.

For each type of solution presented (best, average, and worst) we compare each single result with the best respective one (given in bold) that we were able to find in the literature. The results used have been taken from a number of works as follows: SA [22], LRGA [5], SM[25], GA [21], EP[12] and IPSO[28].

Another important feature of the proposed algorithm is that, as it can be seen in Table 4, the variability of the results is very small. The difference between the worst and best solutions found for each problem is always below 0.3%, while if the best and the average solutions are compared this difference is never larger than 0.11%. This allows for inferring the robustness of the solution since the gaps between the best and

Table 1. Comparison between best results obtained by the BRKGA and the best ones reported in literature

Size	GA	SA	LRGA	EP	IPSO	BRKGA	Ratio	**BRKGA rank**
10	563977	565828	564800	564551	**563954**	563977	100	2nd
20	1125516	1126251	**1122622**	1125494	1125279	1124470	100.16	2nd
40	2249715	2250063	**2242178**	2249093	2248163	2246287	100.18	2nd
60	3375065	–	3371079	3371611	**3370979**	3368542	99.93	1st
80	4505614	4498076	4501844	4498479	**4495032**	4493658	99.97	1st
100	5626514	5617876	**5613127**	5623885	5619284	5614522	100.02	2nd

Table 2. Comparison between average results obtained by the BRKGA and the best averages reported in literature

Size	SA	SM	BRKGA	Ratio	**BRKGA rank**
10	**565988**	566787	563996	99.65	1st
20	**1127955**	1128213	1124753	99.72	1st
40	2252125	**2249589**	2247534	99.91	1st
60	–	**3394830**	3372104	99.33	1st
80	4501156	**4494378**	4495632	100.03	2nd
100	5624301	**5616699**	5616734	100.00	2nd

Table 3. Comparison between worst results obtained by the BRKGA and the best worst ones reported in literature

Size	GA	SA	SM	EP	IPSO	BRKGA	Ratio	**BRKGA rank**
10	565606	566260	567022	566231	**564579**	564028	99.90	1st
20	1128790	1129112	1128403	1129793	**1127643**	1125671	99.83	1st
40	2256824	2254539	**2249589**	2256085	2252117	2248510	99.95	1st
60	3382886	–	3408275	3381012	**3379125**	3379915	100.02	2nd
80	4527847	4503987	**4494439**	4512739	4508943	4499207	100.11	2nd
100	5646529	5628506	**5616900**	5639148	5633021	5619581	100.05	2nd

Table 4. Analysis of the variability of the results and execution time

Size	Best	Average	Worst	$\frac{Av-Best}{Best}\%$	$\frac{Worst-Best}{Best}\%$	St. deviation(%)	Av.Time(s)
10	563977	563996	564028	0.003	0.09	0.003	5.1
20	1124470	1124753	1125671	0.03	0.11	0.03	19.8
40	2246287	2247534	2248510	0.06	0.1	0.08	86.6
60	3368542	3372104	3379915	0.11	0.3	0.12	198.0
80	4493658	4495632	4499207	0.04	0.12	0.06	343.8
100	5614522	5616734	5619581	0.04	0.09	0.05	534.3

the worst solutions are very small. Furthermore our worst solutions, when worse than the best worst solutions reported are always within 0.11% of the latter, see Table 3. This is very important since the industry is reluctant to use methods with high variability as this may lead to poor solutions being used.

The BRKGA has been implemented on Matlab and executed on a Pentium IV Core Duo personal computer T 5200, 1.60GHz and 2.0GB RAM. In table 4 we can see the scaling of the execution time with the system size for the proposed BRKGA. Regarding the computational time no exact comparisons may be done since the values are obtained on different hardware, and on the other hand, some authors only report their computational times graphically, as is the case in [28][1]. However, in Figure 5 it can easily be seen that ours are about the same magnitude of the best execution times as in SA [22].

[1] The computational times of the IPSO in Figure 5 are estimated from the results reported graphically.

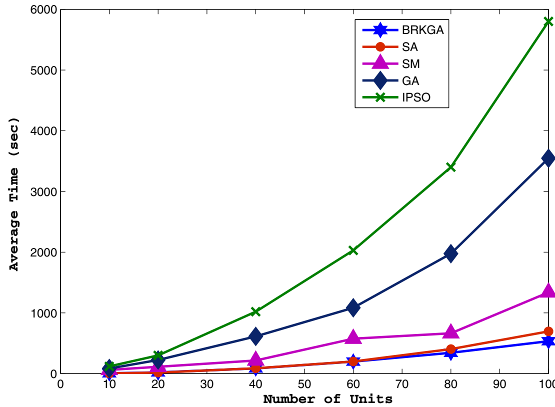


Fig. 5. Average execution time of the different methods

5 Conclusion

A methodology based on a Biased Random Key Genetic Algorithm, following the ideas presented in [9], for finding solutions to the unit commitment problem has been presented. In the solution methodology proposed real valued random keys are used to encode solutions, since they have been proven to perform well in problems where the relative order of tasks is important.

The proposed algorithm was applied to systems with 10, 20, 40, 60, 80, and 100 units with a scheduling horizon of 24 hours. The numerical results have shown the proposed method to improve upon current state of the art, since only for three problems it was not capable of finding better solutions. Furthermore, the results show a further very important feature, lower variability. It should be notice that the difference between the worst and best solutions is always below 0.30%, while the difference between the best and the average solutions is always below 0.11%. This is very important since methods to be used in industrial applications are required to be robust, therefore preventing the use of very low quality solutions.

Acknowledgments. The financial support by FCT, POCI 2010 and FEDER, through project PTDC/EGE-GES/099741/2008 is gratefully acknowledged.

References

1. Abookazemi, K., Mustafa, M.W., Ahmad, H.: Structured Genetic Algorithm Technique for Unit Commitment Problem. International Journal of Recent Trends in Engineering 1(3), 135–139 (2009)
2. Arroyo, J.M., Conejo, A.J.: A parallel repair genetic algorithm to solve the unit commitment problem. IEEE Transactions on Power Systems 17, 1216–1224 (2002)
3. Bean, J.C.: Genetic Algorithms and Random Keys for Sequencing and Optimization. ORSA Journal on Computing 6(2) (1994)

4. Chen, Y.M., Wang, W.S.: Fast solution technique for unit commitment by particle swarm optimisation and genetic algorithm. International Journal of Energy Technology and Policy 5(4), 440–456 (2007)
5. Cheng, C.P., Liu, C.W., Liu, G.C.: Unit commitment by Lagrangian relaxation and genetic algorithms. IEEE Transactions on Power Systems 15, 707–714 (2000)
6. Cohen, A.I., Yoshimura, M.: A Branch-and-Bound Algorithm for Unit Commitment. IEEE Transactions on Power Apparatus and Systems 102, 444–451 (1983)
7. Dudek, G.: Unit commitment by genetic algorithm with specialized search operators. Electric Power Systems Research 72(3), 299–308 (2004)
8. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, New York (1989)
9. Gonçalves, J.F., Resende, M.G.C.: Biased random-key genetic algorithms for combinatorial optimization. Journal of Heuristics (2010), Published online (August 27, 2010). DOI: 10.1007/s10732-010-9143-1
10. Holland, J.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
11. Jenkins, L., Purushothama, G.K.: Simulated annealing with local search-a hybrid algorithm for unit commitment. IEEE Transactions on Power Systems 18(1), 1218–1225 (2003)
12. Juste, K.A., Kita, H., Tanaka, E., Hasegawa, J.: An evolutionary programming solution to the unit commitment problem. IEEE Transactions on Power Systems 14(4), 1452–1459 (1999)
13. Kazarlis, S.A., Bakirtzis, A.G., Petridis, V.: A Genetic Algorithm Solution to the Unit Commitment Problem. IEEE Transactions on Power Systems 11, 83–92 (1996)
14. Lee, F.N.: Short-term unit commitment-a new method. IEEE Transactions on Power Systems 3(2), 421–428 (1998)
15. Merlin, A., Sandrin, P.: A new method for unit commitment at Electricit de France. IEEE Transactions on Power Apparatus Systems 2(3), 1218–1225 (1983)
16. Padhy, N.P.: Unit commitment using hybrid models: a comparative study for dynamic programming, expert system, fuzzy system and genetic algorithms. International Journal of Electrical Power & Energy Systems 23(8), 827–836 (2001)
17. Raglend, I.J., Padhy, N.P.: Comparison of Practical Unit Commitment Solutions. Electric Power Components and Systems 36(8), 844–863 (2008)
18. Rajan, C.C.A., Mohan, M.R.: An evolutionary programming-based tabu search method for solving the unit commitment problem. IEEE Transactions on Power Systems 19(1), 577–585 (2004)
19. Rajan, C.C.A., Mohan, M.R., Manivannan, K.: Refined simulated annealing method for solving unit commitment problem. In: Proceedings of the 2002 International Joint Conference on Neural Networks, IJCNN 2002, vol. 1, pp. 333–338. IEEE, Los Alamitos (2002)
20. Salam, S.: Unit commitment solution methods. Proceedings of World Academy of Science, Engineering and Technology 26, 600–605 (2007)
21. Senjyu, T., Yamashiro, H., Uezato, K., Funabashi, T.: A unit commitment problem by using genetic algorithm based on unit characteristic classification. IEEE Power Engineering Society Winter Meeting 1 (2002)
22. Simopoulos, D.N., Kavatza, S.D., Vournas, C.D.: Unit commitment by an enhanced simulated annealing algorithm. IEEE Transactions on Power Systems 21(1), 68–76 (2006)
23. Sriyanyong, P., Song, Y.H.: Unit commitment using particle swarm optimization combined with Lagrange relaxation. In: Power Engineering Society General Meeting, pp. 2752–2759. IEEE, Los Alamitos (2005)
24. Swarup, K.S., Yamashiro, S.: Unit Commitment Solution Methodology Using Genetic Algorithm. IEEE Transactions on Power Systems 17, 87–91 (2002)

25. Valenzuela, J., Smith, A.E.: A seeded memetic algorithm for large unit commitment problems. Journal of Heuristics 8(2), 173–195 (2002)
26. Viana, A., Sousa, J.P., Matos, M.A.: Fast solutions for UC problems by a new metaheuristic approach. IEEE Electric Power Systems Research 78, 1385–1389 (2008)
27. Xing, W., Wu, F.F.: Genetic algorithm based unit commitment with energy contracts. International Journal of Electrical Power & Energy Systems 24(5), 329–336 (2002)
28. Zhao, B., Guo, C.X., Bai, B.R., Cao, Y.J.: An improved particle swarm optimization algorithm for unit commitment. International Journal of Electrical Power & Energy Systems 28(7), 482–490 (2006)

A Column Generation Approach to Scheduling of Periodic Tasks

Ernst Althaus[1], Rouven Naujoks[2], and Eike Thaden[3]

[1] Johannes Gutenberg-Universität, Mainz,
Max-Planck-Institut für Informatik, Saarbrücken
ernst.althaus@uni-mainz.de, ernst.althaus@mpi-inf.mpg.de
[2] Max-Planck-Institut für Informatik, Saarbrücken
naujoks@mpi-inf.mpg.de
[3] OFFIS, Oldenburg
eike.thaden@offis.de

Abstract. We present an algorithm based on column generation for a real time scheduling problem, in which all tasks appear regularly after a given period. Furthermore, the tasks exchange messages, which have to be transferred over a bus, if the tasks involved are executed on different ECUs. Experiments show that for large instances our preliminary implementation is faster than the previous approach based on an integer linear programming formulation using a state-of-the-art solver.

1 Introduction

We consider the scheduling problem arising in the manufacturing of embedded systems. Besides a set $T = \{t_1, t_2, \ldots, t_l\}$ of tasks specified by their worst-case execution time, deadline and period, we are given a set of messages $M = \{m_1, \ldots, m_x\}$ with a single source task, a set of destination tasks, a transmission time and a deadline. Furthermore we are given a set ECU of identical electronic control units with a certain amount of memory connected over a certain data bus architecture.

We will later discuss several variants of our scheduling problem. Here, we will discuss a specific variant for which we define the feasibility of a schedule.

We assume that the tasks arrive with a fixed rate given as their period. For each task its deadline is smaller or equal to its period which also applies to messages. Each task has to be assigned to exactly one ECU. An ECU can execute exactly one task at a time and uses preemptive fixed-priority scheduling. We assume that we are given deadline monotonic priorities, which was proven to be optimal in our setting (see [3]). A set of tasks is feasible for an ECU, if there is no task whose worst case response time is larger than its deadline. The well-known fix-point equation proposed by Joseph and Pandya in [6] is used to determine the worst case response time. Furthermore, the total memory requirements of the tasks should not exceed the memory of the ECU.

Once the tasks are assigned to ECUs, each message of which at least one target is assigned to another ECU than the source task has to be sent over the

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 340–351, 2011.
© Springer-Verlag Berlin Heidelberg 2011

bus. We assume that all messages for which all targets are on the same ECU as the source meet the deadlines without imposing any restriction. One common bus system is the so called token ring (TAN), in which a token is given to the ECUs in a round robin fashion. Once an ECU has the token, it can send all messages in its outgoing buffer. The worst case delay of a message is attained, if the ECU has just passed the token when the message arrives. In this case, the ECU has to wait for a whole round trip of the token. The delay required for this round trip is called the token rotation time (TRT). The worst possible TRT is attained, if all other messages are send before the message itself, i.e. the sum of the transmission times of all messages that are sent over the bus. Additionally, we assume some fixed delay for passing the token.

Despite the fact that variations of the problem described so far have been subject to scientific research for more than two decades now, it is still of major importance in several industrial sectors, e.g. in aerospace, automotive, automation industries.

The work that is most closely related to ours is the one of Eisenbrand-et al [5], where the authors formulate the problem as an integer linear program which is solved by a standard ILP-solver. We improve upon their work by performing a Dantzig-Wolfe decomposition of their ILP formulation (introducing a column generation approach for several variants of the problem), giving a better running time on large instances.

2 Previous Work

There have been lots of publications in the past tackling the problem of allocating software tasks to processors processors with consideration of scheduling properties. Early works focused on mapping software modules on array or multi processors optimizing communication as in [4] where heuristic algorithms are used. In [9] the problem of finding schedulable deployments is tackled allowing one or more tasks on each ECU (scheduled by a preemptive priority-based scheduler). Here the author used simulated annealing to find semi-optimal solutions. In [2] the author compares several heuristic methods for deployment synthesis but does not discuss exact analysis in detail due the excessive number of combinational solution candidates existing in the design space.

Approaches to find exact solutions have been proposed in [8] (based on SAT solving with binary search on the objective value) and [5] (based on MILP optimization). Both papers focus on the minimization of timing properties on the communication channel. In this paper the second approach is extended using a well-known technique called column generation with the intention to reduce the time required for optimization.

Column generation is a widely used approach to solve hard problems which are decomposed into several local problems that have to be combined to a global optimal solution. In the context of scheduling, the local problem is to assign a subset of the tasks to a particular machine. Hence, column generation is a very natural choice of scheduling problems and several column generation approaches

were already proposed in the literature (see [1] for an overview). The pricing problem is typically to find a set of tasks with maximal profit that fits to a machine, which can often be solved with pseudopolynomial algorithms. In our setting, the problem is rather complicated, as the schedulability of a specific set of tasks is harder to decide because of the periodic appearance of the tasks with different periods. Furthermore, a schedule of the tasks impose a set of messages that have to be routed over a bus and hence a schedule for the messages has to be computed. Depending on the bus architecture, this can be relatively easy or hard to compute.

3 A Column Generation Approach

In this section we will describe our column generation approach for solving the previously defined scheduling problem. We start by formulating the problem as an integer linear program. Given a set $ECU := \{E_1, E_2, \ldots, E_n\}$ of ECUs, we call a subset p of the set of tasks T a *task-assignment pattern* for an $E_j \in ECU$ if the tasks in p can be scheduled on E_j and we denote by $\mathcal{P}^j$ the set of all assignment patterns for E_j. For the ILP formulation we associate with each pattern $p \in \mathcal{P}^j$ a binary variable $x_{j,p}$ indicating whether the pattern is used for the schedule or not. We demand that exactly one pattern is used per ECU, i.e.

$$\forall\, E_j \in ECU\colon \sum_{p \in \mathcal{P}^j} x_{j,p} = 1$$

Furthermore, each task has to be assigned to exactly one ECU, i.e.

$$\forall\, t \in T\colon \sum_{E_j \in ECU} \sum_{p \in \mathcal{P}^j\colon\, t \in p} x_{j,p} = 1$$

Given that M denotes the set of all messages, for $m \in M$ we denote by m_s the source-task of the message m and by m_T be the set of target-tasks of m. Furthermore, for each message, we are given a deadline $d(m)$ and a transmission-time $tt(m)$. Similarly to the sets $\mathcal{P}^j$ we define the set $\mathcal{Q}$. We call a $q \subseteq M$ a *message-pattern* if—given that the messages sent over the bus are exactly the ones in q—all message deadlines are satisfied with respect to some bus protocol. The set $\mathcal{Q}$ is then defined as the set of all message-patterns. A message m is sent over the bus, if at least one of its targets is executed on an ECU different than its source task. As in the case of the task-assignment patterns, we introduce a decision variable z_q for each $q \in \mathcal{Q}$. We demand that we use exactly one pattern, i.e.

$$\sum_{q \in \mathcal{Q}} z_q = 1$$

Furthermore, we introduce indicator variables y_m with y_m being 1 if and only if message m is sent over the bus. Thus, y_m is 0 if and only if all targets of m are

scheduled on the same ECU as the source task which can be established by the constraints

$$\forall\, m \in M\colon y_m = \sum_{E_j \in ECU} \sum_{p \in \mathcal{P}^j\,:\, m_s \in p \,\wedge\, m_T \setminus p \neq \emptyset} x_{j,p}$$

Finally, with the constraints

$$\forall\, m \in M\colon y_m \leq \sum_{q \in \mathcal{Q}\colon m \in q} z_q$$

we ensure that the chosen message-pattern contains all messages that have to be transmitted over the bus. Thus, our scheduling problem can be formulated as follows

$$\begin{aligned}
&\min \quad \sum_{E_j \in ECU} \sum_{p \in \mathcal{P}^j} c(j,p) \cdot x_{j,p} && && (1)\\
\text{st:}\quad & \sum_{E_j \in ECU} \sum_{p \in \mathcal{P}^j} x_{j,p} \cdot p_l = 1 && \forall\, t_l \in T && (2)\\
& \sum_{p \in \mathcal{P}^j} x_{j,p} = 1 && \forall\, e_j \in E && (3)\\
& y_m = \sum_{E_j \in ECU} \sum_{p \in \mathcal{P}^j\,:\, m_s \in p \,\wedge\, m_T \setminus p \neq \emptyset} x_{i,p} && \forall\, m \in M && (4)\\
& y_m \leq \sum_{q \in \mathcal{Q}\colon m \in q} z_q && \forall\, m \in M && (5)\\
& \sum_{q \in \mathcal{Q}} z_q = 1 && && (6)\\
& x_{j,p} \in \mathcal{P}^j && && (7)\\
& z_q \in \mathcal{Q} && && (8)\\
& y_m \in \{0,1\} && && (9)
\end{aligned}$$

Note that in equation 3 p_l is the l-th component of the pattern vector p. In this ILP $c(j,p)$ denotes the cost associated with a task-pattern variable $x_{j,p}$, which depends on the objective function under consideration. We will discuss possible cost functions in Section 3.3 and their impact on the pattern costs.

The main idea for solving this program is to apply a branch and bound approach. For the bounds we will solve the LP-Relaxations of the ILPs occurring in the branching process. Since these ILPs have an exponential number of variables, we will solve them by a column generation approach. In Section 3.2 we will describe our branch and bound step and in Section 3.1 our column generation approach in more detail. In Sections 3.3, 3.4 and 3.5, we discuss several objective functions and the formulations for the sets $\mathcal{P}^j$ and Q and show how the corresponding pricing problems can be solved.

3.1 Solving the Relaxation

By relaxing the integrality constraints of the variables $x_{j,p}$ we have that $x_{j,p} \geq 0$ since the constraints $x_{j,p} \leq 1$ are implied by the constraints (2). Following the *column generation* approach, we start by considering the so-called *master problem* containing at first only a small subset of variables, i.e. all message variables y_m. Since we do not know in general whether our problem is feasible or not, we make it artificially feasible by adding a special ECU, called the *super ECU* E_S which is capable of scheduling all tasks and by allowing only one pattern p_t with $p_t = T$ for E_S. In order to be able to detect whether the original problem is feasible or not, we assign the pattern variable x_{E_S,p_t} a cost that is larger than any optimal solution not using E_S, so that we can easily decide upon this question by inspecting the objective value. Since this choice for $c(E_S, p_t)$ depends on the considered objective function, we postpone its discussion to Section 3.3. Notice that using the super ECU, no message is sent and hence the LP is feasible if it contains an arbitrary message-pattern variable z_M.

Now we solve the master problem by the simplex method and check by solving the so-called *pricing problem* whether there is a non-basic variable $x_{j,p}$ or z_q with negative reduced cost. If such a variable exists, we know—apart from degeneracies—that the optimal solution found for the master problem is not optimal for the original problem and that we have to add this variable to the master problem and to repeat this step. Otherwise, the solution must also be optimal for the original problem and we are done.

We can partition this search for a variable into several subproblems, namely into the search for a variable $x_{j,p}$ for a given ECU E_j and for a variable z_q. Let us start the discussion with the first problem. The column $a_{x_{j,p}}$ in the coefficient matrix of the ILP-relaxation corresponding to a pattern variable $x_{j,p}$ can be written as $(p, I_j, M(p), 0^{|M|}, 0)$ where I_j is a vector in $\{0,1\}^{|ECU|}$ with exactly one 1 at position j and where $M(p)$ contains a 1 for message m, if p implies that m has to be sent over the bus. Thus, by writing d for the vector of dual variables corresponding to the task constraints (2), d' corresponding to the messages constraints (4) and γ corresponding to the constraint (3) of machine j in the master-LP, the pricing problem reads as follows:

$$
\begin{aligned}
&\min \quad c(j,p) - d^T p - d'^T x - \gamma && && (10)\\
\text{subject to:}\quad & x_m \geq p_{m_s} - p_t && \forall\, m \in M, t \in m_T && (11)\\
& p \in \mathcal{P}^j && && (12)\\
& p \in \{0,1\}^{|T|} && && (13)
\end{aligned}
$$

The constraints (11) force x_m to be one, if the source task of the message m is used in the pattern and at least one of the target tasks of m is not used in the pattern. Similarly, we can derive the pricing problem for a variable z_q. In this case, the corresponding column in the coefficient matrix is given by $(0^{|T|}, 0^{|E|}, 0^{|M|}, q, 1)$. With dual variables d'' for the constraints (5) and γ' for (6) the pricing problem reads as follows:

$$\min \quad -d''^T q - \gamma' \tag{14}$$

$$\text{subject to:} \quad q \in \mathcal{Q} \tag{15}$$

We describe the formulations of the constraints (12) and (15) in Section 3.4 and 3.5. For our experiments (see Section 4) we have chosen the so-called TAN-bus architecture which we will explain in more detail in Section 3.5.

3.2 Branch and Bound

The solution of the LP-Relaxation gives us a lower bound on the optimal objective value of the ILP. That is, if the cost of the relaxation is not smaller than the best solution found so far, we can backtrack. Note that if the cost function maps to integral values—as in the case of minimizing the number of used ECUs—we can strengthen this test by rounding the objective value of the LP relaxation up.

Otherwise, we branch by identifying a task t that is mapped on different ECUs and by creating two problems, namely one in which the task is assigned to some specific ECU E_j and one in which it is forbidden to map t on E_j. Fixing or forbidding tasks on certain ECUs translates to fixed pattern variables in the pricing problems, and hence the type of the pricing problem doesn't change.

Note that so far we only obtain upper bounds on the objective value if we find an integral solution in the bounding step. In Section 3.6 we will show how to obtain good upper bounds by solving the problem heuristically in the first place.

3.3 The Objective Functions

There are many ways to define what an optimal scheduling in our setting is. First, we will describe our algorithm in the context of minimizing the number of used ECUs. Then we will alter our algorithm to cope with an objective function proposed by Eisenbrand et al [5].

If we want to minimize the number of ECUs used by a scheduling, we can set for each pattern variable $x_{j,p}$, $c(j,p) = 1$ except for the super ECU where we have to set $c(E_S, p_t) = |ECU| + 1$ to ensure that no solution feasible for the original problem will use the pattern variable x_{E_S,p_t}.

Let us now discuss the modifications necessary to incorporate the objective function of finding a near-equal Processor utilization proposed by Eisenbrand et al. Here, the *utilization* of a pattern p is given by

$$u(p) := \sum_{t_i \in T} p_i \cdot r(t)$$

where $r(t) := w(t)/\tau(t)$. Recall that $w(t)$ is the worst case execution time of task t and $\tau(t)$ is its period. Note that besides trivial input instances, there must be an optimal solution such that all ECU are used. Thus, we can define the *average utilization* of the task system T to be

$$\overline{u}(T) := \frac{1}{|ECU|} \sum_{t \in T} r(t)$$

We can now ask for an assignment of tasks to the ECUs via patterns p_j such that each ECU is assigned exactly one pattern minimizing the function

$$\sum_{e_j \in ECU} |u(p_j) - \overline{u}(T)|$$

where p_j is the pattern assigned to ECU E_j. To incorporate this objective function into our approach we have to assign each pattern variable $x_{j,p}$ the cost $c(j,p) := |u(p) - \overline{u}(T)|$. A solution to the ILP then solves our problem under the new objective function.

Other objective functions are e.g. minimizing the utilization of the bus or keeping one ECU as free as possible to be able to add further tasks on it later.

3.4 Scheduling Periodic Tasks

For the discussion of the scheduling properties (including the periodicity of the problem) of all tasks in the system on their respective target ECUs as determined by choosing patterns from $\mathcal{P}^j$ and the corresponding formulation as an integer linear program we refer to [5] as our formulation used in our implementation is a straightforward adaption of their results. Important variants include the special cases when the deadline is always equal to the period, so-called implicit deadlines and when there are only a small number of different periods. Furthermore, the period can have a jitter, i.e. the tasks appear in fixed periods but can be delayed by some amount.

3.5 The TAN-Bus

Recall that for each message m in M, m_s denotes the source-task of m and that m_T denotes the set of its target-tasks. A message is sent over the bus, if at least one of its targets is executed on another ECU as its source. For each message, we are given a deadline $d(m)$ and a transmission-time $tt(m)$.

First we discuss how one can check the feasibility of a given schedule. For a task t let $E(t)$ be the ECU on which t is scheduled. Let $S = \{m \in M \mid \{E(m_s)\} \neq E(m_T)\}$, where $E(m_T) = \cup_{t \in m_T} E(t)$, be the set of all messages that have to be sent over the bus. The so-called *token rotation time* is then computed as

$$TRT = |E| + \sum_{m \in S} tt(m)$$

It is then required that $d(m) \leq TRT$ for all $m \in S$.

Thus, we can formulate the constraint $q \in \mathcal{Q}$ as

$$\forall\, \overline{m} \in M: \qquad \overline{M}(1 - q_{\overline{m}}) + d(\overline{m}) \geq \sum_{m \in M} tt(m) \cdot q_m \tag{16}$$

where $\overline{M} \geq \sum_{m \in M} tt(m) - d(\overline{m})$. Note that therefore, the pricing problem for the variables z_q is actually a knapsack problem which can be solved very efficiently in a combinatorial way.

An other important type for the bus is the so called CAN-bus, which uses priority routing, i.e. the message with the highest priority is routed first. Deciding whether a set of messages can be send results in a problem similar to the schedulability problem described above. Comparison between TAN-bus and CAN-bus requires further investigation and is out-of-scope of this paper.

3.6 Improvements

In this section we will describe several ideas to increase the performance of our algorithm. We will introduce a technique that allows us to find lower bounds in the bounding step without having to solve the master-LP.

A Lagrangian Bound. We want so solve $min_{Ax=b}c^T x$ with a very large set of variables. We always have a subset of the variables in the LP and iteratively add variables. The question is whether we can obtain bounds on the LP before having priced in all variables. This can be helpful to cut off a subproblem early.

A bound can be computed as follows. Let x' and y' be the primal and dual solution of the current LP, where $x'_v = 0$ for all variables that are not in the current LP. We can compute a bound on the difference of the optimal LP solution for all variables and the current LP solution as follows. $min_{Ax=b}c^T x - c^T x' = min_{Ax=b}(c^T - y'A)x + y'b - c^T x' = min_{Ax=b}(c^T - y'A)x = min_{Ax=b}c_R^T x$ for $c_R^T = c^T - y'A$ being the vector of reduced costs. In our setting $Ax = b$ consists of the constraints (2), (3), (4), (5), (6), (7). We have $min_{(2),(3),(4),(5),(6),(7)} c_R^T x \geq min_{(3),(6),(7)} c_R^T x$. In this systems, the variables for the single ECUs and for the bus are independent, i.e. there is no constraint with non-zero coefficients for variables $x_{i,p}$ and $x_{j,p'}$ for $i \neq j$ from two different ECUs or non-zeros for a variable $x_{i,p}$ and a variable z_q. Hence, the value of this LP is $\sum_j min_{p \in \mathcal{P}^j} c_R(p) + min_{q \in \mathcal{Q}} c_R(q)$. This is the sum of the objective function values of the pricing problems that have to be solved.

Heuristic Solutions. We compute upper bounds on the objective value by computing heuristical solutions in the following way. We start by solving the master-LP and obtain a possibly fractional solution. From this solution we pick the task-pattern $x_{j,p}$ with the highest fractional value and schedule the tasks in p on ECU E_j. Then we eliminate all task-pattern variables which are not compatible with scheduling tasks p on E_j. From the remaining set of pattern variables we again pick the variable with the highest factional value and repeat this procedure until there are no pattern variables left. On this altered problem with an increased set of predeployed tasks we apply our exact algorithm. If the problem turns out to be feasible we have found an integral solution for our problem which can serve as a heuristic solution for our original problem.

Further Improvements. To decrease the number of iterations of the pricing procedure to solve the master-LP, we do not add only one variable to the master-LP, but all variables with negative reduced costs we found in our pricing algorithm.

Notice that the pricing problems according to some given ECU have the same feasible set and differ only in the objective functions. We could observe that it is often the case that functions do not change from one iteration to the other. Therefore, we use a caching mechanism to avoid solving the same pricing problems over and over again.

Furthermore, we noticed that there are many iterations due to the fact that only those constraints of type (4) have non-negative dual values that can't be used in a pattern that seems to be reasonable. In the first iterations these are none, because using the super-ECU is the only feasible solution of the master-LP. Therefore we added a small epsilon to all dual values of the message-constraints, forcing the message-pattern variables to be maximal sets.

4 Experiments

The evaluation of our approach is performed using the following setup. As we adopted concepts from Eisenbrand et al. (see [5]) and extended them, but had no access to the original implementation we partially reimplemented their approach. The constraints for mapping each task to exactly one processor, the response time calculation for each task and the deadline constraints were taken from the paper as is. As communication media we currently only support Token Ring. Both optimization objectives "Near-Equal Processor Utilization" and "Minimizing Number of ECUs" have been reused directly. To have tighter constraints we rely on deadline-monotonic priority assignment, too. Other LP optimizations from [5] have not been adopted.

We aimed to get on the one hand results comparable to former work and on the other hand a large set of random examples.

4.1 Generation of Random Examples

The main parameters for the example generator are the overall number of tasks and the fraction of undeployed tasks. We distinguish predeployed tasks (those tasks have been assigned to an ECU a priori and may not be redeployed) and undeployed tasks (which have not been deployed to any ECU yet). The procedure to build a feasible example is as follows. First the required number of tasks is randomly generated. We assume that all tasks are activated periodically (there is no activation jitter). Signals are generated which connect a sender task with one or more receiver tasks in a tree-like fashion. The result is a set of task/signal trees. Periods of tasks are randomly created for all tasks which are not receiving any signals (root nodes). All other tasks transitively inherit the period of their root node tasks (this applies to signals, too). As for now, each task's deadline is chosen to be equal to its period (applies to signals, too).

In the next step as many ECUs are generated as are required to deploy all task and guarantee a utilization of each ECU of less or equal to 69%. After deploying each task by choosing priorities deadline-monotonic way, according to the Liu and Layland criterion [7] each ECU is schedulable afterwards. Now we perform

a scheduling analysis to calculate for each task (signal) its worst case response time (worst case transmission time). Without violating the schedulability we can now change the deadline D of each task on a ECU iteratively such that it is in the following interval:

$$D_i^{new} \in \left[D_i^{min}, \tau(i)\right], \text{ where} \tag{17}$$

$$D_i^{min} = \max\left\{r_i, \max_{j \in hp(i)} \left\{D_j^{new}\right\}\right\} \tag{18}$$

In these equations D_i^{new} is the to-be-defined deadline of task i, $\tau(i)$ is its period. The $hp(i)$ contains the indices of all higher priority tasks on the same ECU as task i. The new deadline is chosen randomly but with a tendency to be close to the minimal value, which makes the problem harder to solve later on. Note that we preserve the order of task priorities on each ECU by restricting the new deadlines in a way, that they will never violate the deadline-monotonic premise. The set of signals is determined which have to be deployed on the bus because at least on receiver task is on a different ECU than the sender task. The sum of the transmission times of these signals is used as a deadline for all signals in the system. The resulting system will be tighter in terms of scheduling analysis but still be feasible. In the next step a number of tasks depending on the parameter *fraction of undeployed tasks* are randomly chosen and removed from their ECUs. This results in an allocation problem with at least one solution.

4.2 Evaluation

The academic reference model and each generated example were tested with the reimplemented approach by Eisenbrand et al and our new approach (called *CGS*) using the two supported optimization objectives. All experiments were run on one of two compute servers each equipped with four Quad-Core processors (AMD Opteron™ 8378 at 2,4 GHz) and 128 GB RAM. Both implementations were using the commercial LP solver Gurobi Optimizer 4.0. All runtimes are given in consumed CPU time (kernel+user time) of the optimization process.

Table 1 compares the runtimes of our CGS approach and the Eisenbrand approach on the Tindell example. Note the immense runtimes for optimizing the Near-Equal Utilization objective. In our randomly generated examples this could not be reproduced presumably because the bounds of the Tindell example are very tight and therefore optimization is harder.

For the generated examples a timeout of 60min was chosen after which the optimization processes are terminated. We generated and analyzed 773 different

Table 1. Results for Academic Example by Tindell et. al in [9]

Type/Runtime	**CGS**	**Eisenbrand**
Minimize number of ECUs (E)	53 s	126 s
Near-Equal Utilization (U)	205 min	1402 min

Table 2. Results for Generated Examples (type: E=Minimize number of ECUs; U=Near-Equal Utilization)

Nr	#Tasks	#Undepl.	type	Obj. Value	Runtime CGS (s)	Runtime Eis. (s)
1	10	1	E	2	0.03	0.02
2	10	1	U	2	0.03	0.02
3	40	4	U	0.09	0.41	1.7
4	40	4	U	0.09	1.23	1.08
5	100	10	E	13	15.19	28.07
6	120	12	E	18	8.8	60.34
7	100	30	U	0.18	2611.59	329.58
8	30	21	U	0.0002	1121.7	2069.54

examples with number of tasks varying from 10 to 200 and fraction of undeployed tasks between 10% and 90%. For most combinations of these parameters we generated up to 10 examples to increase diversity. For each model we performed optimization types towards both supported objectives with our approach and the Eisenbrand approach for reference. Overall 2234 analyses were performed. In 691 cases the analyses timed out after 60 minutes.

For 545 variants of the input parameters (number of tasks, fraction of undeployed tasks, optimization type) both approaches found feasible solutions. Due to the huge amount of results we show only exemplary data in Table 2. For small systems with up to 40 tasks, CGS rarely outperforms the reference approach (only 10 times out of 325). However times are for most records very close together (difference smaller than 0.1s) making their delta insignificant.

In the range of 40 to 100 tasks CGS gains ground. There are still 174 examples where the Eisenbrand approach is better, but in 30 cases CGS wins. Lines 3 and 4 show that times are still very close together. In the test field of 100 tasks and more, CGS performs better than the reference approach: In 49 cases CGS is significantly faster compared to 41 cases where it is slower. In some rare cases CGS seems to produce long runtimes for the optimization of the Near-Equal Utilization of ECUs as seen in line 7. This has to be further investigated but does not seem to be a common behavior. The Eisenbrand approach suffers from those corner cases, too, as can be seen in line 8.

Increasing the number of tasks comes with increasing complexity for the optimization. Therefore in the range of 100 to 200 tasks, 418 analyses out of 768 timed out. We can not conclude from these results that our approach scales better with respect to the size of the input models than the reference of Eisenbrand. However the results encourage further work in our current direction.

5 Conclusion

We have presented a column generation approach for a scheduling problem with periodic appearance of tasks combined with messages that have to be transfered

over a bus if one of the targets of the message is scheduled on a different ECU than its source.

Our current implementation is faster for large instances than an existing approach based on an integer programming formulation although the pricing problems are currently solved using adapted versions of this formulation.

In future work, we will investigate on combinatorial algorithms for the pricing problems considering some variants of the basic problem that are important in practical applications, e.g. in most real life instances only a small number of different periods occur and the deadline of the tasks are often equal to the period. We believe that using combinatorial algorithms to solve the pricing problems will greatly improve on the running times.

Acknowledgment. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org).

References

1. Akker, M., Hoogeveen, H., Velde, S.: Applying column generation to machine scheduling. In: Desaulniers, G., Desrosiers, J., Solomon, M.M. (eds.) Column Generation, pp. 303–330. Springer, US (2005)
2. Altenbernd, P.: Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks. Ph.D. thesis, University of Paderborn (1996)
3. Baruah, S., Goossens, J.: Scheduling Real-time Tasks: Algorithms and Complexity. In: Handbook of Scheduling: Algorithms, Models, and Performance Analysis, pp. 207–233. Chapman Hall/ CRC Press (2004)
4. Bollinger, S.W., Midkiff, S.F.: Heuristic technique for processor and link assignment in multicomputers. IEEE Trans. Comput. 40, 325–333 (1991)
5. Eisenbrand, F., Damm, W., Metzner, A., Shmonin, G., Wilhelm, R., Winkel, S.: Mapping Task-Graphs on Distributed ECU Networks: Efficient Algorithms for Feasibility and Optimality. In: Proceedings of the 12th IEEE Conference on Embedded and Real-Time Computing Systems and Applications. IEEE Computer Society, Los Alamitos (2006)
6. Joseph, M., Pandya, P.K.: Finding response times in a real-time system. The Computer Journal 29, 390–395 (1986)
7. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM 20, 46–61 (1973)
8. Metzner, A., Fränzle, M., Herde, C., Stierand, I.: An optimal approach to the task allocation problem on hierarchical architectures. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS 2006, p. 178 (2006)
9. Tindell, K., Burns, A., Wellings, A.: Allocating hard real time tasks (an nphard problem made easy). Journal of Real-Time Systems 4, 145–165 (1992)

Fuzzy Clustering the Backward Dynamic Slices of Programs to Identify the Origins of Failure

Saeed Parsa, Farzaneh Zareie, and Mojtaba Vahidi-Asl

Department of Software Engineering, Iran University of Science and Technology, Tehran, Iran
{parsa,m_vahidi_asl}@iust.ac.ir,
farzaneh_zareie@comp.iust.ac.ir

Abstract. In this paper a new technique for identifying the origins of program failure is presented. To achieve this, the outstanding features of both statistical debugging and dynamic slicing techniques are combined. The proposed Fuzzy-Slice technique, computes the full backward dynamic slice of variables used in output statement of a given program in several failing and passing executions. According to the statements presented in the slice of an execution, each run could be converted into an execution point within Euclidean space, namely execution space. Using fuzzy clustering technique, different program execution paths are identified and the fault relevant statements are ranked according to their presence in different clusters. The novel scoring method for identifying fault relevant statements considers the observation of a statement in all execution paths. The promising results on Siemens test suite reveal the high accuracy and precision of the proposed Fuzzy-Slice technique.

Keywords: Software Debugging, Fuzzy Clustering, Backward Dynamic Slicing, Failure.

1 Introduction

No software company could claim that its software products are perfect and bug-free. Even with best-effort of in-house testing teams, software products still ship with undetected latent bugs [15]. A majority of these bugs are encountered in the hands of end users and therefore establishing a distributed crash report feedback systems may help the software companies to take advantage of user community as invaluable volunteer testers [6]. The feedback reports (including failure and successful execution results) collected from a huge number of users could be further analyzed by debuggers to find the cause(s) of program failure [9].

However, the manual analysis of collected data using traditional software debugging techniques is an arduous and inaccurate activity which requires time, effort, and a good understanding of the source code. This has motivated researches to develop automated debugging techniques during past few years [5][6][9][16]. Nevertheless, due to the diverse nature of software faults and complex structure of underlying programs, the process of automating fault localization is not trivial and straightforward.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 352–363, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Although, different techniques and methods have been introduced for software fault localization, there has been an increasing interest in fault localization using statistical methods and program slicing techniques [15]. Program slicing technique first proposed by Weiser [3] computes the static slice of a reference to a given variable at a specific program point. The result is a set of statements statically affecting the value of that variable. To find a software fault, a static slice for variables in a failing output is calculated and presented to the programmer to find and fix the causes of failure. Because of conservative nature of static slicing which considers all static dependences, static slices might be very large as the size of a program grows. As a result, it requires a huge effort of human debugger to inspect the given slice manually in order to detect the real failure cause [2]. To resolve the problem with static slicing technique, dynamic slicing [1] has been proposed which computes all executed statements having influence on a specific execution point. In contrast with static slicing which extracts statements with potentially effect on a given variable, a dynamic slice contains a smaller subset of static slice which have an actual effect on the same variable in a particular execution [1][2][4]. Given a wrong output value, a backward dynamic slice of the value might contain the failure cause that is responsible for producing the failing result [2]. Although, variants of dynamic slicing techniques have been presented in recent years to narrow down the slice size [7][10], they still suffer from some major limitations. Except [10] they have no capability to rank the statements based on their likelihood of being faulty. Therefore, the debugger should examine all statements included in the slice with an equal chance of being faulty. Generally, slicing techniques require only a single failing run for their analysis which seems to be a great advantage when there is no more failing executions available; therefore, they can only identify a fault related to that single failure and they are incapable to detect other unknown faults existed in the program. Furthermore, the computed slice might be too large including too much irrelevant information in addition to the fact that some type of faults (e.g. missing code) cannot be captured by a dynamic slice.

Since statistical debugging techniques rely on larger number of failing and passing executions they can overcome the limitations of slicing techniques [5][6]. They contrast the runtime behavior of correct and incorrect runs to locate faults [9][16]. Typically, the runtime behavior is determined by evaluating simple Boolean expressions called predicates (e.g. directions of branches, the results of function calls, assignment statements) at various program points [9]. To achieve this, an extra code is inserted before each predicate resulted in an instrumented program. The predicate evaluations are gathered from different users which have executed the instrumented programs [6]. A main privilege of statistical techniques is their ability to rank predicates according to their relevance to the faulty code [5]. Another important characteristic of these techniques is their ability to identify faults whose presence may not be known due to considering various failing and passing runs in their analysis [10]. However, these techniques require a large number of passing and failing runs in order to build an appropriate statistical model. Furthermore, due to lightweight instrumentation they may lose their capability to identify the cause of failure if the fault is not located in the predicates [15]. In these cases, they only report those predicates which are highly affected by the fault and the debugger should examine the code manually to find the origin of failure.

To resolve the problem, is this paper a combination of statistical debugging and backward dynamic slicing techniques is performed. The aim is to consider a minimum number of passing and failing runs on one hand and find the cause of failure with less amount of manual code inspection, on the other hand. To achieve this, we require a mechanism that considers different failing execution paths in the program, due to the fact that for a single bug in the program there might be many failing paths which traverse through the faulty statement and result in a failure. The mechanism could convert each execution to an appropriate vector (i.e. execution point) in an execution space and apply an adequate clustering technique on vectors obtained from different failing and passing runs. It is clear that executions in each cluster may share large number of executed statements which result in similar execution paths. However, by using naïve clustering techniques we cannot measure the total belongingness of a particular run in terms of its executed statements (included/not-included in the slice) to different execution paths. The purpose is to find out whether a specific statement is observed in different execution paths and contrast its behavior in failing versus passing runs to identify fault relevant statements.

The proposed Fuzzy-Slice technique in this paper, computes the full backward dynamic slice of each execution of a given program. It, then converts each execution into an execution point within Euclidean space, namely execution space. Using fuzzy clustering technique [11], different execution paths are identified and the fault relevant statements are ranked according to their presence in different clusters by contrasting failing versus passing runs. The novel scoring method for identifying fault relevant statements considers the observation of a statement in all execution paths. To achieve this, for each eligible statement it computes the likelihood of being faulty and the likelihood of being correct and assigns the statement an appropriate score according to the computed likelihood ratios. The high scored statements are then assigned to each cluster and reported to the user as fault suspicious statements. In summary the following contributions have been made in this paper:

1. The program executions are converted into vectors in Euclidean space according to their backward dynamic slices.
2. A fuzzy clustering technique is applied to specify different execution paths in the program.
3. A novel scoring technique for ranking fault relevant statements have been introduced.

The remaining part of this paper is organized as follows. In section 2, an overview of the method including some definitions is described. The experiments and results are shown in section 3. Finally concluding remarks are mentioned in section 4.

2 The Fuzzy-Slice Method

In this section the proposed Fuzzy-Slice method is described in detail. This section is divided into 2 parts. In 2.1 some basic definitions are presented and in 2.2 a description of the proposed method is provided.

2.1 Basic Definitions

In order to combine the idea of backward dynamic slicing technique and statistical debugging, we assume that for each single failure in the program, at least one failing run and a number of passing runs exist. To illustrate the details of the proposed Fuzzy-Slice technique, some basic definitions should be described at first place.

Definition 1. The Dynamic Control Flow Graph for the execution e of program *P*, $\mathrm{DCFG}^{P}e$, is a directed graph containing all executed program statements in the execution e where an edge from node *m* to node *n* depicts that statement *n* is control dependent to statement *m*.

Definition 2. The Dynamic Backward Slice for statement s in the execution e of program *P*, $\mathrm{DBS}^{P}e(s)$, is a set of all statements that affect the value of s because of having data or control dependence with s.

2.2 The Method Overview

The Fuzzy-Slice technique contains two main phases, namely clustering runs according to their similarity and assigning scores to the selected statements. The proposed Fuzzy-Slice method is performed as the following steps:

1) Computing full backward dynamic slice for the failure output of the given program for existing passing and failing test cases.
2) Determining the execution paths using fuzzy clustering method
3) Computing the conditional probability of being faulty and being correct for each eligible program statement
4) Computing the total score of each program statement for each fuzzy cluster and assigning the descending sorted lists of statements to each fuzzy cluster
5) Giving priorities to fuzzy clusters

Each step will be described briefly in the following sub-sections.

2.2.1 Constructing the Execution Points

As mentioned earlier, we assume that we have a number of failing and passing test cases. Imagine that program *P* has an output *op* where it produces incorrect values for some specific test cases. For such *op* the backward dynamic slice is computed for failing and passing test cases. The dynamic slice of variable(s) at *op* includes all those executed statements which actually affect the value of the variable(s) at that point during a particular execution. In other words, a statement belongs to $\mathrm{DBS}^{P}e(op)$ of a variable reference at *op* in execution *e* of program *P*, if there is a chain of dynamic data and/or control dependences from the statement to the variable reference at *op*. With two categories of failing and passing test cases (i.e. input parameters) and the computed backward slices for all existing failing and passing test cases, the aim is to rank the statements presented in the slices based on their relevancy to the program failure. Assume $BDS_{P}^{All}(op)$ contains the union of all slices computed with the existing passing and failing test cases. Therefore, $BDS_{P}^{All}(op)$, contains statements which has been observed in at least one backward dynamic slice in program *P* for

output instance, *op*. Now, we can define our Euclidian execution space where each specific test case has its corresponding execution point in the space.

Definition 3. Given $BDS_P^{All}(op)$, a set of all distinct statements extracted from failing and passing backward dynamic slices for program *P*, we define *Prog-Space*(*P*) as Euclidean space of N-tuples ($y_1, y_2, \ldots, y_N$) where each tuple represents a dimension (also called feature) in the space. Each y_i stands for one and only one specific element in $BDS_P^{All}(op)$ and can only have 3 values (-1, 0, 1). In other words, each statement in $BDS_P^{All}(op)$ constructs a dimension in Prog-Space(*P*). For execution e of program *P*, the value of dimension *s* corresponding to the statement *S* in the execution e considering *op* as incorrect output instance is computed as follows:

1) if $S \in \mathrm{DBS}^P e(op) \rightarrow$ value(s) = 1
2) if $S \notin \mathrm{DBS}^P e(op)$ and $S \in \mathrm{DCFG}^P e \rightarrow$ value(s) = 0
3) if $S \notin \mathrm{DCFG}^P e \rightarrow$ value(s) = -1

With this structure, each program execution is presented as an execution vector (i.e. execution point in the space) according to its executed (included/not included in the slice) and unexecuted statements in that particular run.

2.2.2 Clustering the Execution Points

Now, for each execution, there exist an execution point in the Euclidean space and the aim is to cluster those points based on their distance in order to identify different execution paths (i.e. corresponding to different clusters in the space). However the nature of execution points is such that an execution point is not necessarily belong to only one particular cluster without having relations with other clusters. In other words, an execution point in cluster *c1* may have some common elements with execution points in cluster *c2*. Therefore, instead of naïve clustering technique, it is preferred to apply fuzzy clustering method [11] which is described briefly in the following sub section.

Fuzzy Clustering. In Conventional clustering, a given observation is included in exclusive clusters. Therefore, it is obvious to determinate whether an object belongs to a particular cluster or not and it is not really important to consider the dependence of an object in a cluster with other existing clusters. However, in some situations we require a method that takes into account all possible relations (i.e. similarities) between objects. The fuzzy clustering seems to be applicable in such situations. In fuzzy clustering we face with two factors: membership and membership weight. Suppose that X=($x_1, x_2, \ldots, x_n$) be a given object set of size *n*, the fuzzy cluster for the given set is defined as follows:

$$\mu_k : X \rightarrow [0,1],\ k = 1, 2, \ldots, K. \quad (1)$$

In other words, assuming that we have *K* fuzzy clusters, all elements in the object set are given a value between zero to one which shows the amount that an object belongs to a specific cluster. The weight of belonging degree of object *i* to cluster *k* is denoted as:

$$u_{ik} = \mu_k(x_i), \;\; i = 1,2,\ldots,n, \;\; k = 1,2,\ldots,k. \tag{2}$$

The defined u_{ik} meets the following condition:

$$u_{ik} \in [0,1], \forall i,k; \sum_{k=1}^{K} u_{ik} = 1. \tag{3}$$

A well known fuzzy clustering method is FCM (Fuzzy *C*-Means clustering) which tries to minimize loss function known as the weighted within-class sum of squares. The minimization objective function with the fuzziness ratio *m* for given cluster *k* is computed as follows:

$$J(U) = \sum_{i=1}^{n}\sum_{j=1}^{n}(u_{ik})^m (u_{jk})^m d^2(x_i, x_j), \quad m \in (1, \infty) \tag{4}$$

Where $d^2(x_i,x_j)$ is the Euclidean distance between two objects x_i and x_j. The classifier uses an iterative process to minimize the given objective function. In each iteration, it computes the belonging degrees for objects and *K* new centers of clusters.

2.2.3 Using Fuzzy Clustering to Identify Execution Paths

As mentioned earlier, conventional clustering techniques cannot consider the complete relation of execution paths to each other. In other words, with naïve clustering technique we cannot show how much an execution point corresponding to particular program run in a cluster has similar executed/non-executed statements with program executions in other clusters. The reasoning on which this claim has been based is shown as an example in Figure 1 where execution points are shown with small balls and the centroids are depicted by red balls.

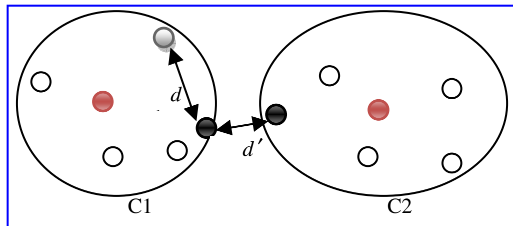


Fig. 1. An example which shows the reason of using Fuzzy Clustering. Although two black points belong to different clusters, they may share common features.

As shown in the Figure, the distance between black and grey execution points in *C1* cluster, *d*, is more than the distance between two black balls, d', in two different clusters *C1* and *C2*. By naive clustering, the similarity between the two black points is not considered while they may share a large number of common statements. Therefore, the semi-similar execution paths cannot be identified, appropriately. Using

FCM clustering method, to each execution point a vector of weights (i.e. belonging degrees) is assigned which shows how much an execution point belongs to different execution paths. Each element *i* of the vector is related to the weight of the object (i.e. execution point) in the associated cluster (i.e. execution path). Since execution points are computed based on dynamic slice of each execution and the points are clustered by fuzzy clustering, we call the technique Fuzzy-Slice.

2.2.4 Computing the Likelihood of Being Faulty/Correct for Statements

The next step is computing the likelihood of being faulty (*LBF*) and the likelihood of being correct (*LBC*) for each statement that is known as a dimension in the execution space. For each cluster we want to estimate the amount that a particular statement has been executed in failing and passing runs. For each existing cluster *k*, the introduced LBC_s^k considers the value of the given statement *s* in each passing execution (i.e passing execution point) in terms of the belonging weight of that point to the cluster. Therefore, for each statement we first compute the value of its corresponding dimension in each passing execution point in addition to the belonging weight of the passing point to the given cluster.

If there are *K* fuzzy clusters, the likelihood of being faulty for the statement *s* in cluster *k* is computed using equation (5):

$$\forall k \in K, N\text{=The number of passing execution points, } LBC_s^k = \frac{\sum_{n=1}^{N} E_n(value(s)) \times \mu_k(E_n)}{N} \tag{5}$$

In a similar way, the likelihood of being faulty for s is computed using equation (6):

$$\forall k \in K, M\text{ =The number of failing execution points, } LBF_s^k = \frac{\sum_{m=1}^{M} E_m(value(s)) \times \mu_k(E_m)}{M} \tag{6}$$

Where E_n in (5) and E_m in (6) denote the execution points corresponding to failing test case *n* and passing test case *m*, respectively.

2.2.5 Assigning Fault Relevance Score to Each Statement

As mentioned earlier, for each statement *s* in given cluster *k* we compute two different ratios: likelihood of being faulty (*LBF*) and likelihood of being correct (*LBC*) according to the membership weight of failing and passing execution points to the corresponding fuzzy cluster, respectively.

It is evident that a given statement with more *LBF* and less *LBC* is more likely to be fault relevant and vice versa. Therefore, the *fault relevance score* (*score* in brief) of statement *s* in cluster *k* is proportional to *LBF* over *LBC* of the statement which could be stated as follows:

$$score_s^k \propto \frac{LBF_s^k}{LBC_s^k} \tag{7}$$

We can rewrite (7) as the following relation:

$$score_s^k = c \times \frac{LBF_s^k}{LBC_s^k}, \text{ } c \text{ is a constant} \tag{8}$$

According to the fact that some statements have similar manner in failing and passing executions, they cannot provide useful information. Hence, we imagine c as the difference between correct and faulty likelihood of the statement s as:

$$c = LBF_s^k - LBC_s^k. \tag{9}$$

Therefore, statements which have been observed equally in failing and passing executions are given less *score* and vise versa. Relation (8) now becomes:

$$score_s^k = (LBF_s^k - LBC_s^k) \times \frac{LBF_s^k}{LBC_s^k} \tag{10}$$

Now we have got an appropriate metric to measure the fault relevance of each statement. The statements are sorted in descending order according to their score in each cluster. First statements with negative *LBF* are eliminated from the list (they have no predictive power). After prioritizing clusters (described in next section) for highest ranked cluster, we investigate whether the statement with the highest score in that cluster is fault relevant. If it is the cause of failure, we report it to the user. Otherwise, we go to the second ranked cluster to examine the highest score statement in that cluster and so on. The process continues until we find the cause of failure or all the clusters are searched with their highest scored statements. In the latter situation, in a hierarchical process the second statement of clusters according to their priority is examined and so on. With this level based strategy, we try not to lose any suspicious statement related to a particular execution path (i.e. cluster)

2.2.6 Prioritizing Clusters for Searching the Failure Cause

An important issue in seeking fault is the order of clusters for which we assign statements. We compute the priority of each cluster according to the percentage of its failed runs. So if we have K clusters, we compute the priority of each as shown blow:

$$\forall k \in K,\ N = The\ Number\ of\ Failed\ Test\ Cases,\ priority_k = \sum_{n=1}^{N} \mu_k(E_n) \tag{11}$$

3 Experimental Results

In this section we compare our work with three outstanding statistical bug localization methods. These techniques are: *Tarantula* [16], *Cooperative Bug isolation* [9] and *SOBER* [5]. Since, slicing techniques do not rank statements we can not assess our work with them. The experiment has been done on Siemens test suite and the obtained results show the success of Fuzzy-Slice in comparison to the named methods. Siemens test suite contains seven middle-sized programs: *Print-tokens, Print-tokens2, Tot-info, Replace, Schedule, Schedule2 and Tcas* [5]. Each program has a number of

faulty versions (i.e. more than 130 versions in total) where in each version at least one semantic fault has been injected. There is a standard test pool including passing and failing test cases for each program. Siemens has been provided by *Software Repository Infrastructure* (SIR) [12].

To compute backward dynamic slices we used the dynamic slicing framework introduced in [10]. The framework instruments a program and executes a gcc compiler to generate binaries and collects the program dynamic data to produce its dynamic dependence graph. The framework contains two main tools: Valgrind [14] and Diablo[13]. The instrumentation is done by Valgrind memory debugger and profiler that also identifies the data dependence among statement execution instances. The Diablo tool is capable to produce control flow graph (i.e. the control dependence among statements) from the generated binaries. Finally we used WET tool [10] to compute backward dynamic slices from an incorrect output value. Since WET does not support floating points, we inevitably exclude tot-info from the experiment.

The important criterion to evaluate a debugging technique is the percentage of code that should be scrutinized manually to reach the main cause of failure. In the rest of this section we talk about how the named methods work and show the result of our experiment in compare with each method.

3.1 Comparison with Tarantula

TARANTULA is a statistical method in bug localization scope [16]. For each statement s , a score namely color is assigned which is based on the number of times the statement is observed in passing over all test cases. The color shows whether the statement is healthy (i.e. no fault relevant). The resulted color follows the relation in equation (12).

$$\text{Color}(s) \propto \frac{\%\,passed(s)}{\%\,passed(s) + \%\,failed(s)} \tag{12}$$

The experimental results on Siemens suite show that Tarantula has detected 68 faults (47%) with less than 10 percentage of manually code inspection where the proposed method has detected 89 faults (74%) with this amount of code inspection. The result of this comparison is presented in Figure 2.

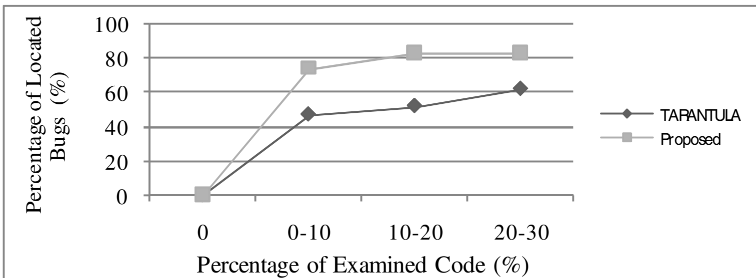


Fig. 2. The result of Tarantula and the proposed method on Siemens suite

3.2 Comparison with Cooperative Bug Isolation

In bug isolation technique proposed by Liblit [9], two conditional probabilities for each predicate p in the program are computed:

$$Context(p)\text{=Probability(program fails} \mid p \text{ is evaluated)} \quad (13)$$
$$Failure(p)\text{=Probability(program fails} \mid p \text{ is evaluated as True)}$$

Liblit ranks the statements according to the difference between the two conditional probabilities. More difference, results in higher fault relevance score. Liblit has detected about 53 faults (32%) with less than 10 percentage of manually code inspection. A comparison with our method is presented in Figure 3.

3.3 Comparison with Sober

The Liblit's approach fails to work when a predicate has been evaluated as *True* in all executions. To resolve the drawback, *SOBER* [5] considers the execution of each predicate as a *Bernoulli* trial with head probability θ. It computes two distributions: f(θ|*passing executions)* and f(θ|*failing executions).* It uses these two distributions to present the evaluation bias of predicate p in each passing and failing executions. Evaluation bias for predicate p, depicted by n_t/n_t+n_f describes the number of times a predicate is evaluated as *True* to the number of times it has been observed (i.e. *True* or *False*) in a specific passing or failing execution. The technique in *SOBER*, applies a null hypothesis based on the equality of variance and median for both failing and passing runs. In cases that there is a high difference between $f(\theta|passing\ executions)$ and $f(\theta|failing\ executions)$ it ranks the predicate as a high bug relevant one. A comparison between *SOBER* and our method is presented in Figure 4.

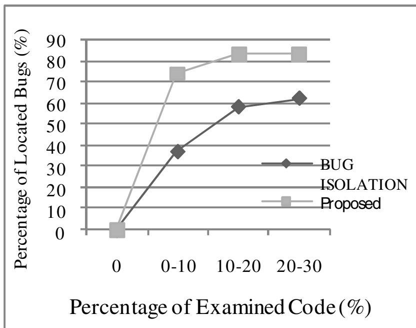


Fig. 3. The result of Bug Isolation (Liblit) and Fuzzy-Slice on Siemens suite

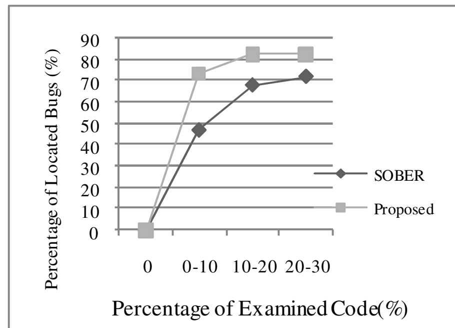


Fig. 4. Comparison of the proposed method with SOBER

3.4 Comparison with Sober, Bug Isolation and Tarantula

Figure 5 shows the comparison of our method with the 3 called methods. As shown in the figure, our method can find 100 bugs in the Siemens test suite (without tot-info) while 18 bugs (18%) among them was located with less than 1% code inspection.

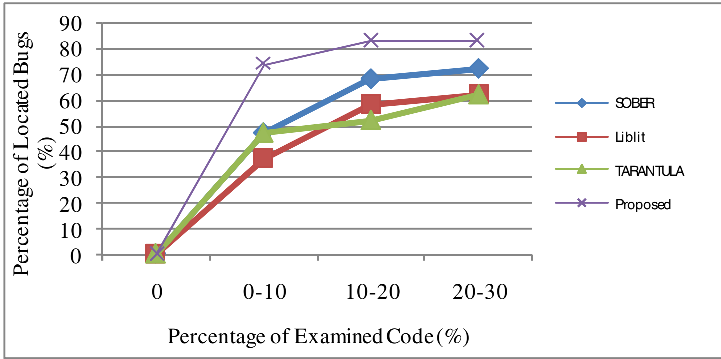


Fig. 5. Comparison of the proposed method with SOBER, Liblit and Tarantula

4 Concluding Remarks and Future Works

In this paper, a new technique for program fault localization is presented. Fuzzy-Slice technique combines the features of full dynamic slicing and statistical methods to locate wider ranges of faults with less amount of code inspection by user. In contrast to statistical fault localization techniques which rely on large number of passing and failing runs, our proposed technique uses fewer numbers of passing runs. The presented execution space helps us to present each execution of a program as a vector in terms of executed statements and computed slices. The Fuzzy clustering method helps to detect different execution paths in the program which results in an effective scoring method. The scoring method takes advantage of fuzzy clusters to rank statements according to their likelihood of being faulty.

For future works, the scalability of the method will be studied and improved. Furthermore, we will seek to find an effective filtering method to reduce the dimension of the program space. For future work, we also study the capability of the proposed method on multiple bug programs.

References

[1] Agrawal, H., Horgan, J.: Dynamic program slicing. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 246–256 (1990)

[2] Agrawal, H., DeMillo, R., Spafford, E.: Debugging with dynamic slicing and backtracking. Software Practice and Experience (SP&E) 23(6), 589–616 (1993)

[3] Weiser, M.: Program slicing. IEEE Transactions on Software Engineering (TSE) SE-10(4), 352–357 (1982)

[4] Zhang, X., Gupta, R., Zhang, Y.: Precise dynamic slicing algorithms. In: IEEE/ACM International Conference on SoftwareEngineering (ICSE), Portland, Oregon, pp. 319–329 (May 2003)

[5] Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.: SOBER: Statistical model-based bug localization. In: Proceedings of the 10th European Software Engineering Conference and 13th ACMSIGSOFT International Symposium on Foundations of Software Engineering, pp. 286–295. ACM Press, New York (2005)

[6] Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 141–154. ACM Press, New York (2003)
[7] Zhang, X., Gupta, N., Gupta, R.: Locating faults through automated predicate switching. In: Proceedings of the International Conference on Software Engineering, Shanghai, China, May 2006, pp. 272–281. ACM Press, New York (2006)
[8] Liblit, B., Naik, M., Zheng, A., Aiken, A., Jordan, M.: Scalable Statistical Bug Isolation. In: Proc. ACM SIGPLAN 2005 Int'l Conf. Programming Language Design and Implementation (PLDI 2005), pp. 15–26 (2005)
[9] Zhang, X., Gupta, N., Gupta, R.: Pruning dynamic slices with confidence. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 169–180 (June 2006)
[10] Sato-Ilic, M., Jain, L.C.: Innovations in Fuzzy Clustering: Theory and Applications. Springer, Heidelberg (2006)
[11] http://www.cse.unl.edu/_galileo/sir
[12] http://www.elis.ugent.be/diablo/
[13] http://valgrind.org/
[14] Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, San Francisco (2006)
[15] Jones, J., Harrold, M.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In: Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng., ASE 2005, pp. 273–282 (2005)

Listing All Maximal Cliques in Large Sparse Real-World Graphs

David Eppstein and Darren Strash

Department of Computer Science, University of California, Irvine, USA

Abstract. We implement a new algorithm for listing all maximal cliques in sparse graphs due to Eppstein, Löffler, and Strash (ISAAC 2010) and analyze its performance on a large corpus of real-world graphs. Our analysis shows that this algorithm is the first to offer a practical solution to listing all maximal cliques in large sparse graphs. All other theoretically-fast algorithms for sparse graphs have been shown to be significantly slower than the algorithm of Tomita et al. (Theoretical Computer Science, 2006) in practice. However, the algorithm of Tomita et al. uses an adjacency matrix, which requires too much space for large sparse graphs. Our new algorithm opens the door for fast analysis of large sparse graphs whose adjacency matrix will not fit into working memory.

Keywords: maximal clique listing, Bron–Kerbosch algorithm, sparse graphs, d-degenerate graphs.

1 Introduction

Clique finding procedures arise in the solutions to a wide variety of important application problems. The problem of finding cliques was first studied in social network analysis, as a way of finding closely-interacting communities of agents in a social network [19]. In bioinformatics, clique finding procedures have been used to find frequently occurring patterns in protein structures [17,26,27], to predict the structures of proteins from their molecular sequences [43], and to find similarities in shapes that may indicate functional relationships between proteins [14]. Other applications of clique finding problems include information retrieval [5], computer vision [20], computational topology [50], and e-commerce [49].

For many applications, we do not want to report one large clique, but all maximal cliques. Any algorithm which solves this problem must take exponential time in the worst-case because graphs can contain an exponential number of cliques [37]. However, graphs with this worst-case behavior are not typically encountered in practice. More than likely, the types of graphs that we will encounter are sparse [16]. Therefore, the feasibility of clique listing algorithms lies in their ability to appropriately handle sparse input graphs. Indeed, it has long been known that certain sparse graph families, such as planar graphs and graphs with low arboricity, contain only a linear number of cliques, and that all maximal cliques in these graphs can be listed in linear time [10,11]. In addition, there are also several methods to list all cliques in time polynomial in the number of cliques reported [46], which can be done faster if parameterized on a sparsity measure such as maximum degree [36].

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 364–375, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Many different clique-finding algorithms have been implemented, and an algorithm of Tomita et al. [45], based on the much earlier Bron–Kerbosch algorithm [8], has been shown through many experiments to be faster by orders of magnitude in practice than others. An unfortunate drawback of the algorithm of Tomita et al., however, is that both its theoretical analysis and implementation rely on the use of an adjacency matrix representation of the input graph. For this reason, their algorithm has limited applicability for large sparse graphs, whose adjacency matrix may not fit into working memory. We therefore seek to have the best of both worlds: we would ideally like an algorithm that rivals the speed of the Tomita et al. result, while having linear storage cost.

Recently, together with Maarten Löffler, the authors developed and published a new algorithm for listing maximal cliques, particularly optimized for the case that the input graph is sparse [13]. This new algorithm combines features of both the algorithm of Tomita et al. and the earlier Bron–Kerbosch algorithm on which it was based, and maintains through its recursive calls a dynamic graph data structure representing the adjacencies between the vertices that remain relevant within each call. When analyzed using parameterized complexity in terms of the degeneracy of the input graph (a measure of its sparsity) its running time is near-optimal in terms of the worst-case number of cliques that a graph with the same sparsity could have. However, the previous work of the authors with Löffler did not include any implementation or experimental results showing the algorithm to be good in practice as well as in theory.

1.1 Our Results

We implement the algorithm of Eppstein, Löffler, and Strash for listing all maximal cliques in sparse graphs [13]. Using a corpus of many large real-world graphs, together with synthetic data including the Moon–Moser graphs as well as random graphs, we compare the performance of our implementation with the algorithm of Tomita et al. We also implement for comparison, a modified version of the Tomita et al. algorithm that uses adjacency lists in place of adjacency matrices, and a simplified version of the Eppstein–Löffler–Strash algorithm that represents its subproblems as lists of vertices instead of as dynamic graphs. Our results show that, for large sparse graphs, the new algorithm is as fast or faster than Tomita et al., and sometimes faster by very large factors. For graphs that are not as sparse, the new algorithm is sometimes slower than the algorithm of Tomita et al., but remains within a small constant factor of its performance.

2 Preliminaries

We work with an undirected graph $G = (V,E)$ with n vertices and m edges. For a vertex v, let $\Gamma(v)$ be its neighborhood $\{w \mid (v,w) \in E\}$, and similarly for a subset $W \subset V$ let $\Gamma(W)$ be the set $\bigcap_{w \in W} \Gamma(w)$, the common neighborhood of all vertices in W.

2.1 Degeneracy

Definition 1 (degeneracy). *The degeneracy of a graph G is the smallest number d such that every subgraph of G contains a vertex of degree at most d.*

```
proc Tomita(P, R, X)
1: if P ∪ X = ∅ then
2:     report R as a maximal clique
3: end if
4: choose a pivot u ∈ P ∪ X to maximize |P ∩ Γ(u)|
5: for each vertex v ∈ P \ Γ(u) do
6:     Tomita(P ∩ Γ(v), R ∪ {v}, X ∩ Γ(v))
7:     P ← P \ {v}
8:     X ← X ∪ {v}
9: end for
```

Fig. 1. The Bron–Kerbosch algorithm with the pivoting strategy of Tomita et al.

Every graph with degeneracy d has a *degeneracy ordering*, a linear ordering of the vertices such that each vertex has at most d neighbors later than it in the ordering. The degeneracy of a given graph and a degeneracy ordering of the graph can both be computed in linear time [6].

2.2 The Algorithm of Tomita et al.

The algorithm of Tomita et al. [45] is an implementation of Bron and Kerbosch's algorithm [8], using a heuristic called *pivoting* [26,9]. The Bron–Kerbosch algorithm is a simple recursive algorithm that maintains three sets of vertices: a partial clique R, a set of candidates for clique expansion P, and a set of forbidden vertices X. In each recursive call, a vertex v from P is added to the partial clique R, and the sets of candidates for expansion and forbidden vertices are restricted to include only neighbors of v. If $P \cup X$ becomes empty, the algorithm reports R as a maximal clique, but if P becomes empty while X is nonempty, the algorithm backtracks without reporting a clique.

In the basic version of the algorithm, $|P|$ recursive calls are made, one for each vertex in P. The pivoting heuristic reduces the number of recursive calls by choosing a vertex u in $P \cup X$ called a *pivot*. All maximal cliques must contain a non-neighbor of u (counting u itself as a non-neighbor), and therefore, the recursive calls can be restricted to the intersection of P with the non-neighbors.

The algorithm of Tomita et al. chooses the pivot so that u has the maximum number of neighbors in P, and therefore the minimum number of non-neighbors, among all possible pivots. Computing both the pivot and the vertex sets for the recursive calls can be done in time $O(|P| \cdot (|P| + |X|))$ within each call to the algorithm, using an adjacency matrix to quickly test the adjacency of pairs of vertices. This pivoting strategy, together with this adjacency-matrix-based method for computing the pivots, leads to a worst-case time bound of $O(3^{n/3})$ for listing all maximal cliques [45].

2.3 The Algorithm of Eppstein, Löffler, and Strash

Eppstein, Löffler, and Strash [13] provide a different variant of the Bron–Kerbosch algorithm that obtains near-optimal worst-case time bounds for graphs with low degeneracy. They first compute a degeneracy ordering of the graph; the outermost call in the

proc Degeneracy(V, E)
1: **for** each vertex v_i in a degeneracy ordering $v_0, v_1, v_2, \ldots$ of (V,E) **do**
2: $\quad P \leftarrow \Gamma(v_i) \cap \{v_{i+1}, \ldots, v_{n-1}\}$
3: $\quad X \leftarrow \Gamma(v_i) \cap \{v_0, \ldots, v_{i-1}\}$
4: $\quad$ Tomita(P, $\{v_i\}$, X)
5: **end for**

Fig. 2. The algorithm of Eppstein, Löffler, and Strash

recursive algorithm selects the vertices v to be used in each recursive call, in this order, without pivoting. Then for each vertex v in the order, a call is made to the algorithm of Tomita et al. [45] to compute all cliques containing v and v's later neighbors, while avoiding v's earlier neighbors. The degeneracy ordering limits the size of P within these recursive calls to be at most d, the degeneracy of the graph.

A simple strategy for determining the pivots in each call to the algorithm of Tomita et al., used as a subroutine within this algorithm, would be to loop over all possible pivots in $X \cup P$ and, for each one, loop over its later neighbors in the degeneracy ordering to determine how many of them are in P. The same strategy can also be used to perform the neighbor intersection required for recursive calls. With the pivot selection and set intersection algorithms implemented in this way, the algorithm would have running time $O(d^2 n 3^{d/3})$, a factor of d larger than the worst-case output size, which is $O(d(n-d)3^{n/3})$.

However, Eppstein et al. provide a refinement of this algorithm that stores, at each level of the recursion, the subgraph of G with vertices in $P \cup X$ and edges having at least one endpoint in P. Using this subgraph, they reduce the pivot computation time to $|P|(|X|+|P|)$, and the neighborhood intersection for each recursive call to time $|P|^2(|X|+|P|)$, which reduces the total running time to $O(dn3^{d/3})$. This running time matches the worst-case output size of the problem whenever $d \leq n - \Omega(n)$. As described by Eppstein et al., storing the subgraphs at each level of the recursion may require as much as $O(dm)$ space. But as we show in Section 3.1, it is possible to achieve the same optimal running time with space overhead $O(n+m)$.

2.4 Tomita et al. with Adjacency Lists

In our experiments, we were only able to run the algorithm of Tomita et al. [45] on graphs of small to moderate size, due to its use of the adjacency matrix representation. In order to have a basis for comparison with this algorithm on larger graphs, we also implemented a simple variant of the algorithm which stores the input graph in an adjacency list representation, and which performs the pivot computation by iterating over all vertices in $P \cup X$ and testing all neighbors for membership in P. When a vertex v is added to R for a recursive call, we can intersect the neighborhood of r with P and X by iterating over its neighbors in the same way.

Let Δ be the maximum degree of the given input graph; then the pivot computation takes time $(O\Delta(|X|+|P|))$. Additionally, preparing subsets for all recursive calls takes time $O(|P|\Delta)$. Fitting these facts into the analysis of Tomita et al. gives us a

$O(\Delta(n-\Delta)3^{\Delta/3})$ time algorithm. Δ may be significantly larger than the degeneracy, so this algorithm's theoretical time bounds are not as good as those of Tomita et al. or Eppstein et al.; nevertheless, the simplicity of this algorithm makes it competitive with the others for many problem instances.

3 Implementation and Experiments

We implemented the algorithm of Tomita et al. using the adjacency matrix representation, and the simple adjacency list representation for comparison. We also implemented three variants of the algorithm of Eppstein, Löffler, and Strash: one with no data structuring, using the fact that vertices have few later neighbors in the degeneracy ordering, an implementation of the dynamic graph data structure that only uses $O(m+n)$ extra space total, and an alternative implementation of the data structure based on bit vectors. The bit vector implementation executed no faster than the data structure implementation, so we omit its experimental timings and any discussion of its implementation details.

3.1 Implementation Details

We maintain the sets of vertices P and X in a single array, which is passed between recursive calls. Initially, the array contains the elements of X followed by the elements of P. We keep a reverse lookup table, so that we can look up the index of a vertex in constant time. With this lookup table, we can tell whether a vertex is in P or X in constant time, by testing that its index is in the appropriate subarray. When a vertex v is added to R in preparation for a recursive call, we reorder the array. Vertices in $\Gamma(v)\cap X$ are moved to the end of the X subarray, and vertices in $\Gamma(v)\cap P$ are moved to the beginning of the P subarray (see Figure 3). We then make a recursive call on the subarray containing the vertices $\Gamma(v)\cap(X\cup P)$. After the recursive call, we move v to X by swapping it to the beginning of the P subarray and moving the boundary so that v is in the X subarray. Of course, moving vertices between sets will affect P and X in higher recursive calls. Therefore, in a given recursive call, we maintain a list of the vertices that are moved from P to X, and move these vertices back to P when the call ends.

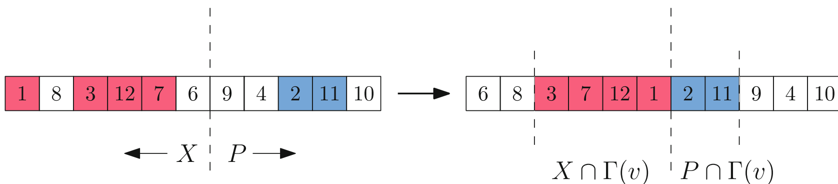


Fig. 3. When a vertex v is added to the partial clique R, its neighbors in P and X (highlighted in this example) are moved toward the dividing line in preparation for the next recursive call

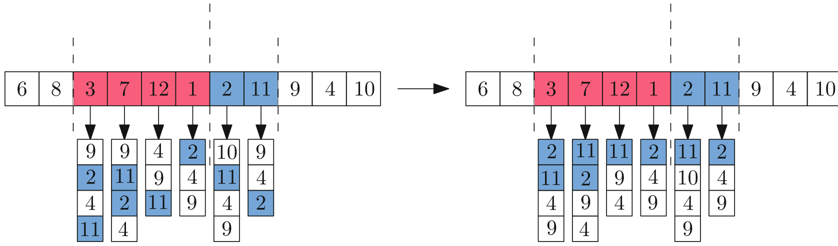


Fig. 4. For each vertex in $P \cup X$, we keep an array containing neighbors in P. We update these arrays whenever a vertex is moved from P to R, and whenever we need to intersect a neighborhood with P and X for a recursive call.

The pivot computation data structure is stored as set of arrays, one for each potential pivot vertex u in $P \cup X$, containing the neighbors of u in P. Whenever P changes, we reorder the elements in these arrays so that neighbors in P are stored first (see Figure 4). Computing the pivot is as simple as iterating through each array until we encounter a neighbor that is not in P. This reordering procedure allows us to maintain one set of arrays throughout all recursive calls, requiring linear space total. Making a new copy of this data structure for each recursive call would require space $O(dm)$.

3.2 Results

We implemented all algorithms in the C programming language, and ran experiments on a Linux workstation running the 32-bit version of Ubuntu 10.10, with a 2.53 GHz Intel Core i5 M460 processor (with three cache levels of 128KB, 512KB, and 3,072KB respectively) and 2.6GB of memory. We compiled our code with version 4.4.5 of the gcc compiler with the `-O2` optimization flag.

In our tables of results, “tomita” is the algorithm of Tomita et al., “maxdegree” is the simple implementation of Tomita et al.’s algorithm for adjacency lists, and “hybrid” and “degen” are the implementations of Eppstein, Löffler, and Strash with no data structure and with the linear space data structure, respectively. We provide the elapsed running running times (in seconds) for each of these algorithms; an asterisk indicates that the algorithm was unable to run on that problem instance due to time or space limitations. In addition, we list the number of vertices n, edges m, the degeneracy d, and the number of maximal cliques μ.

Our primary experimental data consisted of four publicly-available databases of real-world networks, including non-electronic and electronic social networks as well as networks from bioinformatics applications.

– A data base curated by Mark Newman [40] (Table 1) which consists primarily of social networks; it also includes word co-occurrence data and a biological neural network. Many of its graphs were too small for us to time our algorithms accurately, but our algorithm was faster than that of Tomita et al. on all four of the largest graphs; in one case it was faster by a factor of approximately 130.

- The BioGRID data [44] (Table 2) consists of several protein-protein interaction networks with from one to several thousand vertices, and varying sparsities. Our algorithm was significantly faster than that of Tomita et al. on the worm and fruitfly networks, and matched or came close to its performance on all the other networks, even the relatively dense yeast network.
- We also tested six large social and bibliographic networks that appeared in the Pajek data set but were not in the other data sets [7] (Table 3). Our algorithm was consistently faster on these networks. Due to their large size, the algorithm of Tomita et al. was unable to run on two of these networks; nevertheless, our algorithm found all cliques quickly in these graphs.
- We also tested a representative sample of graphs from the Stanford Large Network Dataset Collection [32] (Table 4). These included road networks, a co-purchasing network from Amazon.com data, social networks, email networks, a citation network, and two Web graphs. Nearly all of these input graphs were too large for the Tomita et al. algorithm to fit into memory. For graphs which are extremely sparse, it is no surprise that the maxdegree algorithm was faster than our algorithm, but our algorithm was consistently fast on each of these data sets, whereas the maxdegree algorithm was orders of magnitude slower than our algorithm on the large soc-wiki-Talk network.

Table 1. Experimental results for Mark Newman's data sets [40]

graph	n	m	d	μ	tomita	maxdegree	hybrid	degen
zachary [48]	34	78	4	39	< 0.01	< 0.01	< 0.01	< 0.01
dolphins [35]	62	159	4	84	< 0.01	< 0.01	< 0.01	< 0.01
power [47]	4,941	6,594	5	5,687	0.29	< 0.01	0.01	0.01
polbooks [28]	105	441	6	199	< 0.01	< 0.01	< 0.01	< 0.01
adjnoun [29]	112	425	6	303	< 0.01	< 0.01	< 0.01	< 0.01
football [15]	115	613	8	281	< 0.01	< 0.01	< 0.01	< 0.01
lesmis [25]	77	254	9	59	< 0.01	< 0.01	< 0.01	< 0.01
celegensneural [47]	297	1,248	9	856	< 0.01	< 0.01	< 0.01	< 0.01
netscience [39]	1,589	2,742	19	741	0.02	< 0.01	< 0.01	< 0.01
internet [40]	22,963	48,421	25	39,275	6.68	0.28	0.11	0.11
condmat-2005 [38]	40,421	175,693	29	34,274	39.65	0.22	0.32	0.35
polblogs [4]	1,490	16,715	36	49,884	0.08	0.28	0.18	0.12
astro-ph [38]	16,706	121,251	56	15,794	3.44	0.19	0.22	0.23

Table 2. Experimental results for BioGRID data sets (PPI Networks)[44]

graph	n	m	d	μ	tomita	maxdegree	hybrid	degen
mouse	1,455	1,636	6	1,523	0.01	< 0.01	< 0.01	< 0.01
worm	3,518	3,518	10	5,652	0.14	0.01	0.01	0.01
plant	1,745	3,098	12	2,302	0.02	< 0.01	< 0.01	< 0.01
fruitfly	7,282	24,894	12	21,995	0.62	0.03	0.03	0.04
human	9,527	31,182	12	23,863	1.06	0.03	0.05	0.05
fission-yeast	2,031	12,637	34	28,520	0.06	0.12	0.09	0.07
yeast	6,008	156,945	64	738,613	1.74	11.37	4.22	2.17

Table 3. Experimental results for Pajek data sets [7]

graph	n	m	d	μ	tomita	maxdegree	hybrid	degen
foldoc [21]	13,356	91,471	12	39,590	2.16	0.11	0.14	0.13
eatRS [23]	23,219	304,937	34	298,164	7.62	1.52	1.55	1.03
hep-th [3]	27,240	341,923	37	446,852	12.56	3.40	2.40	1.70
patents [18]	240,547	560,943	24	482,538	*	0.56	1.22	1.65
days-all [12]	13,308	148,035	73	2,173,772	5.83	62.86	9.94	5.18
ND-www [1]	325,729	1,090,108	155	495,947	*	1.80	1.81	2.12

Table 4. Experimental results for Stanford data sets [32]

graph	n	m	d	μ	tomita	maxdegree	hybrid	degen
roadNet-CA [34]	1,965,206	2,766,607	3	2,537,996	*	2.00	5.34	5.81
roadNet-PA [34]	1,088,092	1,541,898	3	1,413,391	*	1.09	2.95	3.21
roadNet-TX [34]	1,379,917	1,921,660	3	1,763,318	*	1.35	3.72	4.00
amazon0601 [30]	403,394	2,443,408	10	1,023,572	*	3.59	5.01	6.03
email-EuAll [31]	265,214	364,481	37	265,214	*	4.93	1.25	1.33
email-Enron [24]	36,692	183,831	43	226,859	31.96	2.78	1.30	0.90
web-Google [2]	875,713	4,322,051	44	1,417,580	*	9.01	8.43	9.70
soc-wiki-Vote [33]	7,115	100,762	53	459,002	0.96	4.21	2.10	1.14
soc-slashdot0902 [34]	82,168	504,230	55	890,041	*	7.81	4.20	2.58
cit-Patents [18]	3,774,768	16,518,947	64	14,787,032	*	28.56	49.22	58.64
soc-Epinions1 [42]	75,888	405,740	67	1,775,074	*	27.87	9.24	4.78
soc-wiki-Talk [33]	2,394,385	4,659,565	131	86,333,306	*	$> 18,000$	542.28	216.00
web-berkstan [34]	685,231	6,649,470	201	3,405,813	*	76.90	31.81	20.87

Table 5. Experimental results for Moon–Moser [37] and DIMACS benchmark graphs [22]

Graphs	n	m	d	μ	tomita	maxdegee	hybrid	degen
M-M-30	30	405	27	59,049	0.04	0.04	0.06	0.04
M-M-45	45	945	42	14,348,907	7.50	15.11	20.36	10.21
M-M-48	48	1080	45	43,046,721	22.52	48.37	63.07	30.22
M-M-51	51	1224	48	129,140,163	67.28	150.02	198.06	91.80
MANN_a9	45	918	27	590,887	0.44	0.88	0.90	0.53
brock_200_2	200	9876	84	431,586	0.55	2.95	2.61	1.22
c-fat200-5	200	8473	83	7	0.01	0.01	0.01	0.01
c-fat500-10	500	46627	185	8	0.04	0.04	0.09	0.12
hamming6-2	64	1824	57	1,281,402	1.36	4.22	4.15	2.28
hamming6-4	64	704	22	464	< 0.01	< 0.01	< 0.01	< 0.01
johnson8-4-4	70	1855	53	114,690	0.13	0.35	0.40	0.24
johnson16-2-4	120	5460	91	2,027,025	5.97	27.05	31.04	12.17
keller4	171	9435	102	10,284,321	5.98	24.97	26.09	11.53
p_hat300-1	300	10933	49	58,176	0.07	0.29	0.25	0.15
p_hat300-2	300	21928	98	79,917,408	91.31	869.34	371.72	163.16

As a reference point, we also ran our experimental comparisons using the two sets of graphs that Tomita et al. used in their experiments. First, Tomita et al. used a data set from a DIMACS challenge, a collection of graphs that were intended as difficult examples for clique-finding algorithms, and that have been algorithmically generated (Table 5). And second, they generated graphs randomly with varying edge densities; in order to replicate their results we generated another set of random graphs with the same parameters (Table 6). The algorithm of Eppstein, Löffler, and Strash runs about 2 to 3 times slower than that of Tomita et al. on many of these graphs; this confirms that the algorithm is still competitive on graphs that are not sparse, in contrast to the

Table 6. Experimental results on random graphs

Graphs		d	μ	tomita	maxdegree	hybrid	degen
n	p						
100	0.6	51	59,898	0.08	0.26	0.25	0.14
	0.7	59	439,928	0.50	2.04	1.85	0.99
	0.8	70	5,776,276	6.29	28.00	24.86	11.74
	0.9	81	240,998,654	249.15	1136.15	1028.84	425.85
300	0.1	21	3,663	< 0.01	0.01	0.01	< 0.01
	0.2	47	18,911	0.02	0.07	0.08	0.05
	0.3	74	86,179	0.10	0.44	0.49	0.24
	0.4	101	555,724	0.70	4.24	3.97	1.67
	0.5	130	4,151,668	5.59	42.37	36.35	13.05
	0.6	162	72,454,791	101.35	958.74	755.86	227.00
500	0.1	39	15,311	0.02	0.03	0.06	0.04
	0.2	81	98,875	0.11	0.46	0.61	0.27
	0.3	127	701,292	0.86	5.90	6.10	2.29
	0.5	225	103,686,974	151.67	1888.20	1521.90	375.23
700	0.1	56	38,139	0.04	0.10	0.19	0.09
	0.2	117	321,245	0.37	2.01	2.69	1.00
	0.3	184	3,107,208	4.06	36.13	38.12	11.47
1,000	0.1	82	99,561	0.11	0.34	0.70	0.28
	0.2	172	1,190,899	1.45	10.35	14.48	4.33
	0.3	266	15,671,489	21.96	262.64	280.58	66.05
2,000	0.1	170	750,991	1.05	5.18	11.77	3.13
3,000	0.1	263	2,886,628	4.23	27.51	68.52	13.62
10,000	0.001	7	49,716	1.19	0.04	0.07	0.07
	0.003	21	141,865	1.30	0.11	0.36	0.26
	0.005	38	215,477	1.47	0.25	1.03	0.51
	0.01	80	349,244	2.20	1.01	5.71	1.66
	0.03	262	3,733,699	9.96	20.66	133.94	20.67

competitors in Tomita et al.'s paper which ran 10 to 160 times slower on these input graphs. The largest of the random graphs in the second data set were generated with edge probabilities that made them significantly sparser than the rest of the set; for those graphs our algorithm outperformed that of Tomita et al by a factor that was as large as 30 on the sparsest of the graphs. The maxdegree algorithm was even faster than our algorithm in these cases, but it was significantly slower on other data.

4 Conclusion

We have shown that the algorithm of Eppstein, Löffler, and Strash is a practical algorithm for large sparse graphs. This algorithm is highly competitive with the algorithm of Tomita et al. on sparse graphs, and within a small constant factor on other graphs. The advantage of this algorithm is that it requires only linear space for storing the graph and all data structures. It does not suffer from the drawback of requiring an adjacency matrix, which may not fit into memory. Its closest competitor in this respect, the Tomita et al. algorithm modified to use adjacency lists, is sometimes faster by a small factor but is also sometimes slower by a large factor. Thus, the algorithm of Eppstein et al. is a fast and reliable choice for listing maximal cliques, especially when the input graphs are large and sparse.

For future work, it would be interesting to compare our results with those of other popular clique listing algorithms. We attempted to include results from Patric Östergård's popular Cliquer program [41] in our tables; however, at the time of writing, its newly implemented functionality for listing all maximal cliques returns incorrect results.

Acknowledgments. We thank Etsuji Tomita and Takeaki Uno for helpful discussions. This research was supported in part by the National Science Foundation under grant 0830403, and by the Office of Naval Research under MURI grant N00014-08-1-1015.

References

1. Self-organized networks database, University of Notre Dame
2. Google programming contest (2002), `http://www.google.com/programming-contest/`
3. Kdd cup (2003), `http://www.cs.cornell.edu/projects/kddcup/index.html`
4. Adamic, L.A., Glance, N.: The political blogosphere and the 2004 us election. In: Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005)
5. Augustson, J.G., Minker, J.: An analysis of some graph theoretical cluster techniques. J. ACM 17(4), 571–588 (1970)
6. Batagelj, V., Zaveršnik, M.: An $O(m)$ algorithm for cores decomposition of networks (2003), `http://arxiv.org/abs/cs.DS/0310049`
7. Batagelj, V., Mrvar, A.: Pajek datasets (2006), `http://vlado.fmf.uni-lj.si/pub/networks/data/`
8. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. Commun. ACM 16(9), 575–577 (1973)

9. Cazals, F., Karande, C.: A note on the problem of reporting maximal cliques. Theor. Comput. Sci. 407(1-3), 564–568 (2008)
10. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. 14(1), 210–223 (1985)
11. Chrobak, M., Eppstein, D.: Planar orientations with low out-degree and compaction of adjacency matrices. Theor. Comput. Sci. 86(2), 243–266 (1991)
12. Corman, S.R., Kuhn, T., Mcphee, R.D., Dooley, K.J.: Studying complex discursive systems: Centering resonance analysis of communication. Human Communication Research 28(2), 157–206 (2002)
13. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in sparse graphs in near-optimal time. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010. LNCS, vol. 6506, pp. 403–414. Springer, Heidelberg (2010)
14. Gardiner, E.J., Willett, P., Artymiuk, P.J.: Graph-theoretic techniques for macromolecular docking. J. Chem. Inf. Comput. Sci. 40(2), 273–279 (2000)
15. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proc. Natl. Acad. Sci. USA 99, 7821–7826 (2002)
16. Goel, G., Gustedt, J.: Bounded arboricity to determine the local structure of sparse graphs. In: Fomin, F.V. (ed.) WG 2006. LNCS, vol. 4271, pp. 159–167. Springer, Heidelberg (2006)
17. Grindley, H.M., Artymiuk, P.J., Rice, D.W., Willett, P.: Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. J. Mol. Biol. 229(3), 707–721 (1993)
18. Hall, B.H., Jaffe, A.B., Trajtenberg, M.: The NBER patent citation data file: Lessons, insights and methodological tools. Tech. rep. (2001), NBER Working Paper 8498
19. Harary, F., Ross, I.C.: A procedure for clique detection using the group matrix. Sociometry 20(3), 205–215 (1957)
20. Horaud, R., Skordas, T.: Stereo correspondence through feature grouping and maximal cliques. IEEE Trans. Patt. An. Mach. Int. 11(11), 1168–1180 (1989)
21. Howe, D.: Foldoc: Free on-line dictionary of computing, `http://foldoc.org/`
22. Johnson, D.J., Trick, M.A. (eds.): Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13. American Mathematical Society, Boston (1996)
23. Kiss, G., Armstrong, C., Milroy, R., Piper, J.: An associative thesaurus of English and its computer analysis. In: Aitken, A.J., Bailey, R., Hamilton-Smith, N. (eds.) The Computer and Literary Studies, University Press, Edinburgh (1973)
24. Klimt, B., Yang, Y.: Introducing the enron corpus. In: CEAS 2004: Proceedings of the 1st Conference on Email and Anti-Spam (2004)
25. Knuth, D.E.: The Stanford GraphBase: A Platform for Combinatorial Computing. Addison-Wesley, Reading (1993)
26. Koch, I.: Enumerating all connected maximal common subgraphs in two graphs. Theor. Comput. Sci. 250(1-2), 1–30 (2001)
27. Koch, I., Lengauer, T., Wanke, E.: An algorithm for finding maximal common subtopologies in a set of protein structures. J. Comput. Biol. 3(2), 289–306 (1996)
28. Krebs, V.: `http://www.orgnet.com/` (unpublished)
29. Leicht, E.A., Holme, P., Newman, M.E.J.: Vertex similarity in networks. Phys. Rev. E 73 (2006)
30. Leskovec, J., Adamic, L., Adamic, B.: The dynamics of viral marketing. ACM Transactions on the Web 1(1) (2007)
31. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graph evolution: Densification and shrinking diameters. ACM Transactions on Knowledge Discovery from Data 1(1) (2007)
32. Leskovec, J.: Stanford large network dataset collection, `http://snap.stanford.edu/data/index.html`

33. Leskovec, J., Huttenlocher, D., Kleinberg, J.: Predicting positive and negative links in online social networks. In: Proc. 19th Int. Conf. on World Wide Web, WWW 2010, pp. 641–650. ACM, New York (2010)
34. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics 6(1), 29–123 (2009)
35. Lusseau, D., Schneider, K., Boisseau, O.J., Haase, P., Slooten, E., Dawson, S.M.: The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. Behavioral Ecology and Sociobiology 54, 396–405 (2003)
36. Makino, K., Uno, T.: New algorithms for enumerating all maximal cliques. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 260–272. Springer, Heidelberg (2004)
37. Moon, J.W., Moser, L.: On cliques in graphs. Israel J. Math. 3(1), 23–28 (1965)
38. Newman, M.E.J.: The structure of scientific collaboration networks. Proc. Natl. Acad. Sci. USA 98, 404–409 (2001)
39. Newman, M.E.J.: Finding community structure in networks using the eigenvectors of matrices. Phys. Rev. E 74(3), 36104 (2006)
40. Newman, M.E.J.: http://www-personal.umich.edu/~mejn/netdata/
41. Niskanen, S., Östergård, P.R.J.: Cliquer user's guide, version 1.0. Tech. Rep. T48, Helsinki University of Technology (2003)
42. Richardson, M., Agrawal, R., Domingos, P.: Trust management for the semantic web. In: ISWC (2003)
43. Samudrala, R., Moult, J.: A graph-theoretic algorithm for comparative modeling of protein structure. J. Mol. Biol. 279(1), 287–302 (1998)
44. Stark, C., Breitkreutz, B.J., Reguly, T., Boucher, L., Breitkreutz, A., Tyers, M.: BioGRID: a general repository for interaction datasets. Nucleic Acids Res. 34, 535–539 (2006)
45. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. Theor. Comput. Sci. 363(1), 28–42 (2006)
46. Tsukiyama, S., Ide, M., Ariyoshi, H., Shirakawa, I.: A new algorithm for generating all the maximal independent sets. SIAM J. Comput. 6(3), 505–517 (1977)
47. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature 393, 440–442 (1998)
48. Zachary, W.W.: An information flow model for conflict and fission in small groups. J. Anthropol. Res. 33, 452–473 (1977)
49. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In: Proc. 3rd Int. Conf. Knowledge Discovery and Data Mining, pp. 283–286. AAAI Press, Menlo Park (1997), http://www.aaai.org/Papers/KDD/1997/KDD97-060.pdf
50. Zomorodian, A.: The tidy set: a minimal simplicial set for computing homology of clique complexes. In: Proc. 26th ACM Symp. Computational Geometry, pp. 257–266 (2010), http://www.cs.dartmouth.edu/~afra/papers/socg10/tidy-socg.pdf

Customizable Route Planning

Daniel Delling[1], Andrew V. Goldberg[1], Thomas Pajor[2,⋆], and Renato F. Werneck[1]

[1] Microsoft Research Silicon Valley
{dadellin,goldberg,renatow}@microsoft.com
[2] Karlsruhe Institute of Technology
pajor@kit.edu

Abstract. We present an algorithm to compute shortest paths on continental road networks with arbitrary metrics (cost functions). The approach supports turn costs, enables real-time queries, and can incorporate a new metric in a few seconds—fast enough to support real-time traffic updates and personalized optimization functions. The amount of metric-specific data is a small fraction of the graph itself, which allows us to maintain several metrics in memory simultaneously.

1 Introduction

The past decade has seen a great deal of research on finding point-to-point shortest paths on road networks [7]. Although Dijkstra's algorithm [10] runs in almost linear time with very little overhead, it still takes a few seconds on continental-sized graphs. Practical algorithms use a two-stage approach: *preprocessing* takes a few minutes (or even hours) and produces a (linear) amount of auxiliary data, which is then used to perform *queries* in real time. Most previous research focused on the most natural metric, driving times. Real-world systems, however, often support other natural metrics as well, such as shortest distance, walking, biking, avoid U-turns, avoid/prefer freeways, or avoid left turns.

We consider the *customizable route planning* problem, whose goal is to perform real-time queries on road networks with *arbitrary metrics*. Such algorithms can be used in two scenarios: they may keep several active metrics at once (to answer queries for any of them), or new metrics can be generated on the fly. A system with these properties has obvious attractions. It supports real-time traffic updates and other dynamic scenarios, allows easy customization by handling any combination of standard metrics, and can even provide personalized driving directions (for example, for a truck with height and weight restrictions). To implement such a system, we need an algorithm that allows real-time queries, has fast customization (a few seconds), and keeps very little data for each metric. Most importantly, it must be *robust*: all three properties must hold for *any metric*. No existing algorithm meets these requirements.

To achieve these goals, we distinguish between two features of road networks. The *topology* is a set of static properties of each road segment or turn, such as

⋆ This work was done while the third author was at Microsoft Research Silicon Valley.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 376–387, 2011.
© Springer-Verlag Berlin Heidelberg 2011

physical length, road category, speed limits, and turn types. The *metric* encodes the actual cost of traversing a road segment or taking a turn. It can often be described compactly, as a function that maps (in constant time) the properties of an edge/turn into a cost. We assume the topology is shared by the metrics and rarely changes, while metrics may change quite often and even coexist.

To exploit this separation, we consider algorithms for customizable route planning with *three stages*. The first, *metric-independent preprocessing*, may be relatively slow, since it is run infrequently. It takes only the graph topology as input, and may produce a fair amount of auxiliary data (comparable to the input size). The second stage, *metric customization*, is run once for each metric, and must be much quicker (a few seconds) and produce little data—a small fraction of the original graph. Finally, the *query stage* uses the outputs of the first two stages and must be fast enough for real-time applications.

In Section 2 we explore the design space by analyzing the applicability of existing algorithms to this setting. We note that methods with a strong hierarchical component, the fastest in many situations, are too sensitive to metric changes. We focus on separator-based methods, which are more robust but have often been neglected in recent research, since published results made them seem uncompetitive: the highest speedups over Dijkstra observed were lower than 60 [17], compared to thousands or millions with other methods.

Section 3 revisits and thoroughly reengineers a separator-based algorithm. By applying existing acceleration techniques, recent advances in graph partitioning, and some engineering effort, we can answer queries on continental road networks in about a millisecond, with much less customization time (a few seconds) and space (a few tens of megabytes) than existing acceleration techniques.

Another contribution of our paper is a careful treatment of turn costs (Section 4). It has been widely believed that any algorithm can be easily augmented to handle these efficiently, but we note that some methods actually have a significant performance penalty, especially if turns are represented space-efficiently. In contrast, we can handle turns naturally, with little effect on performance.

We stress that our algorithms are not meant to be the fastest on any particular metric. For "nice" metrics, our queries are somewhat slower than the best hierarchical methods. However, our queries are robust and suitable for real-time applications with arbitrary metrics, including those for which the hierarchical methods fail. Our method can quickly process new metrics, and the metric-specific information is small enough to keep several metrics in memory at once.

2 Previous Techniques

There has been previous work on variants of the route planning problem that deal with multiple metrics in a nontrivial way. The preprocessing of SHARC [3] can be modified to handle multiple (known) metrics at once. In the *flexible routing problem* [11], one must answer queries on linear combinations of a small set of metrics (typically two) known in advance. Queries in the *constrained routing problem* [23] must avoid entire classes of edges. In multi-criteria optimization [8], one must find Pareto-optimal paths among multiple metrics. ALT [14] and CH [12] can adapt

to small changes in a benign base metric without rerunning preprocessing in full. All these approaches must know the base metrics in advance, and for good performance the metrics must be few, well-behaved, and similar to one another. In practice, even seemingly small changes to the metric (such as higher U-turn costs) render some approaches impractical. In contrast, we must process metrics as they come, and assume nothing about them.

We now discuss the properties of existing point-to-point algorithms to determine how well they fit our design goals. Some of the most successful existing methods—such as reach-based routing [15], contraction hierarchies (CH) [12], SHARC [3], transit node routing [2], and hub labels [1]—rely on the strong *hierarchy* of road networks with travel times: the fastest paths between faraway regions of the graph tend to use the same major roads.

For metrics with strong hierarchies, such as travel times, CH has many of the features we want. During preprocessing, CH heuristically sorts the vertices in increasing order of importance, and *shortcuts* them in this order. (To *shortcut* v, we temporarily remove it from the graph and add arcs as necessary to preserve the distances between its neighbors.) Queries run bidirectional Dijkstra, but only follow arcs or shortcuts to more important vertices. If a metric changes only slightly, one can keep the order and recompute the shortcuts in about a minute [12]. Unfortunately, an order that works for one metric may not work for a substantially different one (e.g., travel times and distances, or a major traffic jam). Furthermore, queries are much slower on metrics with less-pronounced hierarchies [4]. More crucially, the preprocessing stage can become impractical (in terms of space and time) for bad metrics, as Section 4 will show.

In contrast, techniques based on *goal direction*, such as PCD [21], ALT [14], and arc flags [16], produce the same amount of auxiliary data for any metric. Queries are not robust, however: they can be as slow as Dijkstra for bad metrics. Even for travel times, PCD and ALT are not competitive with other methods.

A third approach is based on *graph separators* [17,18,19,25]. During preprocessing, one computes a multilevel partition of the graph to create a series of interconnected overlay graphs. A query starts at the lowest (local) level and moves to higher (global) levels as it progresses. These techniques predate hierarchy-based methods, but their query times are widely regarded as uncompetitive in practice, and they have not been tested on continental-sized road networks. (The exceptions are recent extended variants [6,22] that achieve great query times by adding many more edges during preprocessing, which is costly in time and space.) Because preprocessing and query times are essentially metric-independent, separator-based methods are the most natural fit for our problem.

3 Our Approach

We will first describe a basic algorithm, then consider several techniques to make it more practical, using experimental results to guide our design. Our code is written in C++ (with OpenMP for parallelization) and compiled with Microsoft Visual C++ 2010. We use 4-heaps as priority queues. Experiments were run on a commodity workstation with an Intel Core-i7 920 (four cores clocked at

2.67 GHz and 6 GB of DDR3-1066 RAM) running Windows Server 2008 R2. Our standard benchmark instance is the European road network, with 18 million vertices and 42 million arcs, made available by PTV AG for the 9th DIMACS Implementation Challenge [9]. Vertex IDs and arc costs are both 32-bit integers.

We must minimize *metric customization time*, *metric-dependent space* (excluding the original graph), and *query time*, while keep metric-independent time and space reasonable. We evaluate our algorithms on 10 000 s–t queries with s and t picked uniformly at random. We focus on finding shortest path *costs*; Section 4 shows how to retrieve the actual paths. We report results for travel times and travel distances, but *by design* our algorithms work well for any metric.

Basic Algorithm. Our *metric-independent preprocessing* stage partitions the graph into connected cells with at most U (an input parameter) vertices each, with as few boundary arcs (arcs with endpoints in different cells) as possible.

The *metric customization* stage builds a graph H containing all boundary vertices (those with at least one neighbor in another cell) and boundary arcs of G. It also contains a *clique* for each cell C: for every pair (v, w) of boundary vertices in C, we create an arc (v, w) whose cost is the same as the shortest path (restricted to C) between v and w (or infinite if w is not reachable from v). We do so by running Dijkstra from each boundary vertex. Note that H is an *overlay* [24]: the distance between any two vertices in H is the same as in G.

Finally, to perform a *query* between s and t, we run a bidirectional version of Dijkstra's algorithm on the graph consisting of the union of H, C_s, and C_t. (Here C_v denotes the subgraph of G induced by the vertices in the cell containing v.)

As already mentioned, this is the basic strategy of separator-based methods. In particular, HiTi [19] uses edge-based separators and cliques to represent each cell. Unfortunately, HiTi has not been tested on large road networks; experiments were limited to small grids, and the original proof of concept does not appear to have been optimized using modern algorithm engineering techniques.

Our first improvement over HiTi and similar algorithms is to use PUNCH [5] to partition the graph. Recently developed to deal with road networks, it routinely finds solutions with half as many boundary edges (or fewer), compared to the general-purpose partitioners (such as METIS [20]) commonly used by previous algorithms. Better partitions reduce customization time and space, leading to faster queries. For our experiments, we used relatively long runs of PUNCH, taking about an hour. Our results would not change much if we used the basic version of PUNCH, which is only about 5% worse but runs in mere minutes.

We use parallelism: queries run forward and reverse searches on two CPU cores, and customization uses all four (each cell is processed independently).

Sparsification. Using full cliques in the overlay graph may seem wasteful, particularly for well-behaved metrics. At the cost of making its topology metric-dependent, we consider various techniques to reduce the overlay graph.

The first approach is *edge reduction* [24], which eliminates clique arcs that are not shortest paths. After computing all cliques, we run Dijkstra from each vertex v in H, stopping as soon as all neighbors of v (in H) are scanned. Note that these searches are usually quick, since they only visit the overlay.

A more aggressive technique is to preserve some internal cell vertices [6,17,25]. If $B = \{v_1, v_2, \ldots, v_k\}$ is the set of boundary vertices of a cell, let T_i be the shortest path tree (restricted to the cell) rooted at v_i, and let T_i' be the subtree of T_i consisting of the vertices with descendants in B. We take the union $C = \cup_{i=1}^{k} T_i'$ of these subtrees, and shortcut all internal vertices with two neighbors or fewer. Note that this *skeleton graph* is technically not an overlay, but it preserves distances between all *boundary* vertices, which is what we need.

Finally, we tried a lightweight *contraction* scheme. Starting from the skeleton graph, we greedily shortcut low-degree internal vertices, stopping when no such operation is possible without increasing the number of edges by more than one.

Fig. 1 (left) compares all four overlays (cliques, reduced cliques, skeleton, and CH-skeleton) on travel times and travel distances. Each plot relates the total query time and the amount of metric-independent data for different values of U (the cell size). Unsurprisingly, all overlays need more space as the number of cells increases (i.e., as U decreases). Query times, however, are minimized when the effort spent on each level is balanced, which happens for $U \approx 2^{15}$.

To analyze preprocessing times (not depicted in the plots), take $U = 2^{15}$ (with travel times) as an example. Finding full cliques takes only 40.8 s, but edge reduction (45.8 s) or building the skeleton graph (45.1 s) are almost as cheap. CH-skeleton, at 79.4 s, is significantly more expensive, but still practical. Most methods get faster as U gets smaller: full cliques take less than 5 s with $U = 256$. The exception is CH-skeleton: when U is very small, the combined size of all skeletons is quite large, and processing them takes minutes.

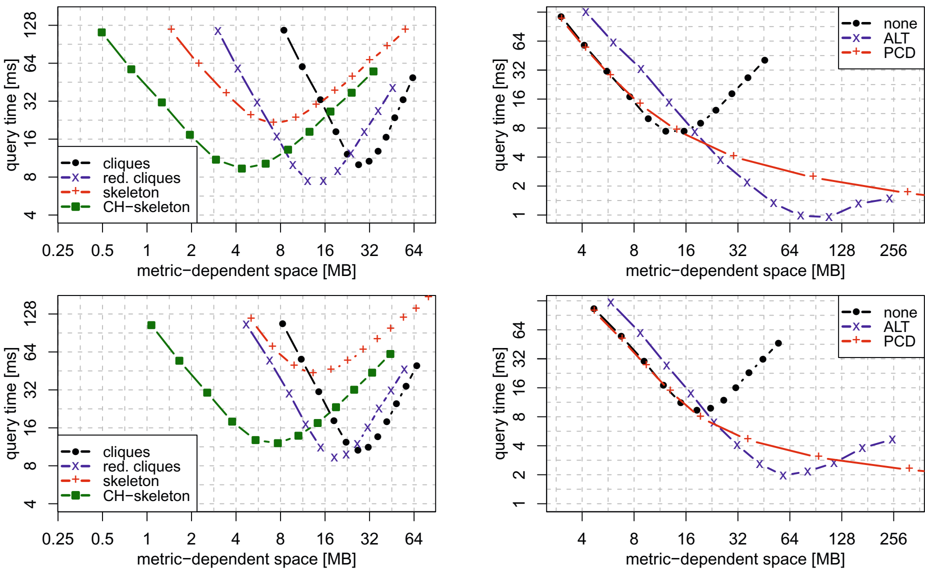


Fig. 1. Effect of sparsification (left) and goal direction (right) for travel times (top) and distances (bottom). The i-th point from the left indicates $U = 2^{20-i}$.

In terms of query times and metric-dependent space, however, CH-skeleton dominates pure skeleton graphs. Decreasing the number of edges (from 1.2M with reduced cliques to 0.8M with skeletons, for $U = 2^{15}$ with travel times) may not be enough to offset an increase in the number of vertices (from 34K to 280K), to which Dijkstra-based algorithms are more sensitive. This also explains why reduced cliques yield the fastest queries, with full cliques not far behind.

All overlays have worse performance when we switch from travel times to distances (with less pronounced hierarchies), except full cliques. Since edge reduction is relatively fast, we use reduced cliques as the default overlay.

Goal-direction. For even faster queries, we can apply more sophisticated techniques (than bidirectional Dijkstra) to search the overlay graph. While in principle any method could be used, our model restricts us to those with metric-independent preprocessing times. We tested PCD and ALT.

To use PCD (Precomputed Cluster Distances) [21] with our basic algorithm, we do the following. Let k be the number of cells found during the metric independent preprocessing ($k \approx n/U$). During metric customization, we run Dijkstra's algorithm k times on the overlay graph to compute a $k \times k$ matrix with the distances between all cells. Queries then use the matrix to guide the bidirectional search by pruning vertices that are far from the shortest path. Note that, unlike "pure" PCD, we use the overlay graph during customization and queries.

Another technique is *core ALT* (CALT) [4]. Queries start with bidirectional Dijkstra searches restricted to the source and target cells. Their boundary vertices are then used as starting points for an ALT (A^* search/ landmarks/triangle inequality) query on the overlay graph. The ALT preprocessing runs Dijkstra $O(L)$ times to pick L vertices as landmarks, and stores distances between these landmarks and all vertices in the overlay. Queries use these distances and the triangle inequality to guide the search towards the goal. A complication of core-based approaches [15,4] is the need to pick nearby overlay vertices as *proxies* for the source or target to get their distance bounds. Hence, queries use four CPU cores: two pick the proxies, while two conduct the actual bidirectional search.

Fig. 1 (right) shows the query times and the metric-dependent space consumption for the basic algorithm, CALT (with 32 *avoid* landmarks [15]), and PCD, with reduced cliques as overlay graphs. With some increase in space, both goal-direction techniques yield significantly faster queries (around one millisecond). PCD, however, needs much smaller cells, and thus more space and customization time (about a minute for $U = 2^{14}$) than ALT (less than 3 s). Both methods are more effective for travel times than travel distances.

Multiple Levels. To accelerate queries, we can use multiple levels of overlay graphs, a common technique for partition-based approaches, including HiTi [19]. We need *nested partitions* of G, in which every boundary edge at level i is also a boundary edge at level $i-1$, for $i > 1$. The level-0 partition is the original graph, with each vertex as a cell. For the i-th level partition, we create a graph H_i as before: it includes all boundary arcs, plus an overlay linking the boundary vertices within a cell. Note that H_i can be computed using only H_{i-1}. We use PUNCH to create multilevel partitions, in top-down fashion.

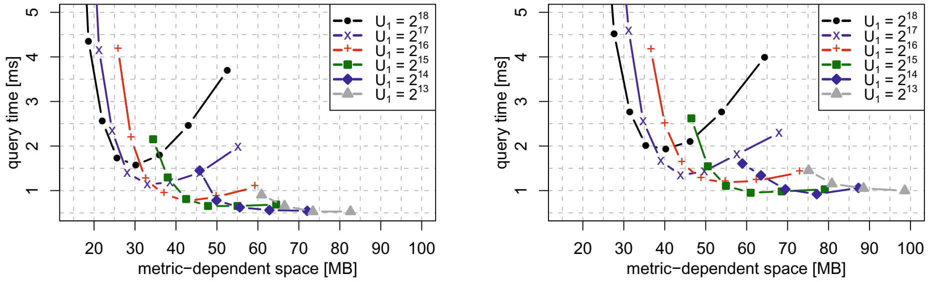


Fig. 2. Performance of 2-level CALT with travel times (left) and distances (right). For each line, U_1 is fixed and U_0 varies; the i-th point from the right indicates $U_0 = 2^{7+i}$.

An s–t query runs bidirectional Dijkstra on a restricted graph G_{st}. An arc (v, w) from H_i will be in G_{st} if both v and w are in the same cell as s or t at level $i + 1$. Goal-direction can still be used on the top level.

Fig. 2 shows the performance of the multilevel algorithm with two overlay levels (with reduced cliques) and ALT on the top level. We report query times and metric-dependent space for multiple values of U_0 and U_1, the maximum cell sizes on the bottom and top levels. A comparison with Fig. 1 reveals that using two levels enables much faster queries for the same space. For travel times, a query takes 1 ms with about 40 MB (with $U_0 = 2^{11}$ and $U_1 = 2^{16}$). Here it takes 16 s to compute the bottom overlay, 5 s to compute the top overlay, and only 0.5 s to process landmarks. With 60 MB, queries take as little as 0.5 ms.

Streamlined Implementation. Although sparsification techniques save space and goal direction accelerates queries, the improvements are moderate and come at the expense of preprocessing time, implementation complexity, and metric-independence (the overlay topology is only metric-independent with full cliques). Furthermore, the time and space requirements of the simple clique implementation can be improved by representing each cell of the partition as a *matrix*, making the performance difference even smaller. The matrix contains the 32-bit distances among its entry and exit vertices (these are the vertices with at least one incoming or outgoing boundary arc, respectively; most boundary vertices are both). We also need arrays to associate rows (and columns) with the corresponding vertex IDs, but these are small and shared by all metrics.

We thus created a matrix-based *streamlined implementation* that is about twice as fast as the adjacency-based clique implementation. It does not use edge reduction, since it no longer saves space, slows down customization, and its effectiveness depends on the metric. (Skipping infinite matrix entries would make queries only slightly faster.) Similarly, we excluded CALT from the streamlined representation, since its queries are complicated and have high variance [4].

Customization times are typically dominated by building the overlay of the lowest level, since it works on the underlying graph directly (higher levels work on the much smaller cliques of the level below). As we have observed, smaller cells tend to lead to faster preprocessing. Therefore, as an optimization, the

Table 1. Performance of various algorithms for travel times and distances

	travel times				travel distances			
	CUSTOMIZING		QUERIES		CUSTOMIZING		QUERIES	
algorithm [cell sizes]	time [s]	space [MB]	vertex scans	time [ms]	time [s]	space [MB]	vertex scans	time [ms]
CALT [2^{11}:2^{16}]	21.3	37.1	5292	0.92	17.2	48.9	5739	1.26
MLD-1 [2^{14}]	4.9	10.1	45420	5.81	4.8	10.1	47417	6.12
MLD-2 [2^{12}:2^{18}]	5.0	18.8	12683	1.82	5.0	18.8	13071	1.83
MLD-3 [2^{10}:2^{15}:2^{20}]	5.2	32.7	6099	0.91	5.1	32.7	6344	0.98
MLD-4 [2^{8}:2^{12}:2^{16}:2^{20}]	4.7	59.5	3828	0.72	4.7	59.5	4033	0.79
CH economical	178.4	151.3	383	0.12	1256.9	182.5	1382	1.33
CH generous	355.6	122.8	376	0.10	1987.4	165.8	1354	1.29

streamlined implementation includes a *phantom level* (with $U = 32$) to accelerate customization, but throws it away for queries, keeping space usage unaffected. For MLD-1 and MLD-2, we use a second phantom level with $U = 256$ as well.

Table 1 compares our streamlined multilevel implementation (called MLD, with up to 4 levels) with the original 2-level implementation of CALT. For each algorithm, we report the cell size bounds in each level. (Because CALT accelerates the top level, it uses different cell sizes than MLD-2.) We also consider two versions of CH: the first (*economical*) minimizes preprocessing times, and the second (*generous*) the number of shortcuts. For CH, we report the total space required to store the shortcuts (8 bytes per arc, excluding the original graph). For all algorithms, preprocessing uses four cores and queries use at least two.

We do not permute vertices after CH preprocessing (as is customary to improve query locality), since this prevents different metrics from sharing the same graph. Even so, with travel times, CH queries are one order of magnitude faster than our algorithm. For travel distances, MLD-3 and MLD-4 are faster than CH, but only slightly. For practical purposes, all variants have fast enough queries.

The main attraction of our approach is efficient metric customization. We require much less space: for example, MLD-2 needs about 20 MB, which is less than 5% of the original graph and an order of magnitude less than CH. Most notably, customization times are small. We need only 5 seconds to deal with a new metric, which is fast enough to enable personalized driving directions. This is two orders of magnitude faster than CH, even for a well-behaved metric. Phantom levels help here: without them, MLD-1 would need about 20 s.

Note that CH customization can be faster if the processing order is fixed in advance [12]. The economical variant can rebuild the hierarchy (sequentially) in 54 s for travel times and 178 s for distances (still slower than our method). Unfortunately, using the order for one metric to rebuild another is only efficient if they are very similar [11]. Also note that one can save space by storing only the upper part of the hierarchy [7], at the expense of query times.

Table 1 shows that we can easily deal with real-time traffic: if all edge costs change (due to a traffic update), we can handle new queries after only 5 seconds. We can also support *local updates* quite efficiently. If a single edge cost changes,

we must recompute at most one cell on each level, and MLD-4 takes less than a millisecond to do so. This is another reason for not using edge reduction or CALT: with either technique, changes in one cell may propagate beyond it.

4 Turns

So far, we have considered a simplified (but standard [7]) representation of road networks, with each intersection corresponding to a single vertex. This is not very realistic, since it does not account for turn costs (or restrictions, a special case). Of course, any algorithm can handle turns simply by working on an expanded graph. A traditional [7] representation is *arc-based*: each vertex represents one *exit point* of an intersection, and each arc is a road segment followed by a turn.

This is wasteful. We propose a *compact representation* in which each intersection becomes a single vertex with some associated information. If a vertex u has p incoming and q outgoing arcs, we associate a $p \times q$ *turn table* T_u to it, where $T_u[i, j]$ represents the turn from the ith incoming arc into the jth outgoing arc[1]. In addition, we store with each arc (v, w) its *tail order* (its position among v's outgoing arcs) and its *head order* (its position among w's incoming arcs). These orders may be arbitrary. Since degrees are small, 4 bits for each suffice.

In practice, many vertices share the same turn table. The total number of such *intersection types* is modest—in the thousands rather than millions. For example, many degree-four vertices in the United States have four-way stop signs. Each distinct turn table is thus stored only once, and each vertex keeps a pointer to the appropriate type, with little overhead.

Dijkstra's algorithm, however, becomes more complicated. In particular, it may now visit each vertex (intersection) multiple times, once for each entry point. It essentially simulates an execution on the arc-based expanded representation, which increases its running time on Europe from 3 s to about 12 s. With a *stalling* technique, we can reduce the time to around 7 s. When scanning one entry point of an intersection, we can set bounds for its other entry points, which are not scanned unless their own distance labels are smaller than the bounds. These bounds depend on the turn table, and can be computed during customization.

To support the compact representation, MLD needs two minor changes. First, it uses turn-aware Dijkstra on the lowest level (but not on higher ones). Second, matrices in each cell now represent paths between incoming and outgoing *boundary arcs* (and not boundary vertices, as before). The difference is subtle. With turns, the distance from a boundary vertex v to an exit point depends on whether we enter the cell from arc (u, v) or arc (w, v), so each arc needs its own entry in the matrix. Since most boundary vertices have only one incoming (and outgoing) boundary arc, the matrices are only slightly larger.

We are not aware of publicly-available realistic turn data, so we augment our standard benchmark instance. For every vertex v, we add a turn between each incoming and each outgoing arc. A turn from (u, v) to (v, w) is either a *U-turn*

[1] In our customizable setting, each entry should represent just a turn type (such as "left turn with stop sign"), since its cost may vary with different metrics.

Table 2. Performance of various algorithms on Europe with varying U-turn costs

	U-turn: 1 s				U-turn: 100 s			
	CUSTOMIZING		QUERIES		CUSTOMIZING		QUERIES	
algorithm	time [s]	space [MB]	vertex scans	time [ms]	time [s]	space [MB]	vertex scans	time [ms]
MLD-1 [2^{14}]	5.9	10.5	44832	9.96	7.5	10.5	62746	12.43
MLD-2 [2^{12}:2^{18}]	6.3	19.2	12413	3.07	8.4	19.2	16849	3.55
MLD-3 [2^{10}:2^{15}:2^{20}]	7.3	33.5	5812	1.56	9.2	33.5	6896	1.88
MLD-4 [2^{8}:2^{12}:2^{16}:2^{20}]	5.8	61.7	3556	1.18	7.5	61.7	3813	1.28
CH expanded	3407.4	880.6	550	0.18	5799.2	931.1	597	0.21
CH compact	846.0	132.5	905	0.19	23774.8	304.0	5585	2.11

(if $u = w$) or a *standard turn* (if $u \neq w$), and each of these two types has a cost. We have not tried to further distinguish between turn types, since any automated method would not reflect real-life turns. However, adding U-turn costs is enough to reproduce the key issue we found on realistic (proprietary) data.

Table 2 compares some algorithms on Europe augmented with turns. We consider two metrics, with U-turn costs set to 1 s or 100 s. The metrics are otherwise identical: arc costs represent travel times and standard turns have zero cost. We tested four variants of MLD (with one to four levels) and two versions of CH (generous): *CH expanded* is the standard algorithm run on the arc-based expanded graph, while *CH compact* is modified to run on the compact representation. Column *vertex scans* counts the number of heap extractions.

Small U-turn costs do not change the shortest path structure of the graph much. Indeed, CH compact still works quite well: preprocessing is only three times slower (than reported in Table 1), the number of shortcuts created is about the same, and queries take marginally longer. Using higher U-turn costs (as in a system that avoids U-turns), however, makes preprocessing much less practical. Customization takes more than 6 hours, and space more than doubles. Intuitively, nontrivial U-turn costs are harder to handle because they increase the importance of certain vertices; for example, driving around the block may become a shortest path. Query times also increase, but are still practical. (Note that recent independent work [13] shows that additional tuning can make compact CH somewhat more resilient: changing U-turn costs from zero to 100 s increases customization time by a factor of only two. Unfortunately, forbidding U-turns altogether still slows it down by an extra factor of 6).

With the expanded representation, CH preprocessing is much costlier when U-turns are cheap (since it runs on a larger graph), but is much less sensitive to an increase in the U-turn cost; queries are much faster as well. The difference in behavior is justified. While the compact representation forces CH to assign the same "importance" (order) to different entry points of an intersection, the expanded representation lets it separate them appropriately.

MLD is much less sensitive to turn costs. Compared to Table 1, we observe that preprocessing space is essentially the same (as expected). Preprocessing and query times increase slightly, mainly due to the lower level: high U-turn

costs decrease the effectiveness of the stalling technique on the turn-enhanced graph.

In the most realistic setting, with nontrivial U-turn costs, customization takes less than 10 seconds on our commodity workstation. This is more than enough to handle frequent traffic updates, for example. If even more speed is required, one could simply use more cores: speedups are almost perfect. On a server with two 6-core Xeon 5680 CPUs running at 3.33 GHz, MLD-4 takes only 2.4 seconds, which is faster than just running sequential Dijkstra on this input.

Path Unpacking. So far, we have reported the time to compute only the distance between two points. Following the parent pointers of the meeting vertex of forward and backward searches, we may obtain a path containing shortcuts. To unpack a level-i shortcut, we run bidirectional Dijkstra on level $i-1$ (and recurse as necessary). Using all 4 cores, unpacking less than doubles query times, with no additional customization space. (In contrast, standard CH unpacking stores the "middle" vertex of every shortcut, increasing the metric-dependent space by 50%.) For even faster unpacking, one can store a bit with each arc at level i indicating whether it appears in a shortcut at level $i+1$. This makes unpacking 4 times faster for MLD-2, but has little effect on MLD-3 and MLD-4.

5 Conclusion

Recent advances in graph partitioning motivated us to reexamine the separator-based multilevel approach to the shortest path problem. With careful engineering, we drastically improved query speedups relative to Dijkstra from less than 60 [17] to more than 3000. With turn costs, the speedup increases even more, to 7000. This makes real-time queries possible. Furthermore, by explicitly separating metric customization from graph partitioning, we enable new metrics to be processed in a few seconds. The result is a flexible and practical solution to many real-life variants of the problem. It should be straightforward to adapt it to augmented scenarios, such as mobile or time-dependent implementations. (In particular, a unidirectional version of MLD is also practical.) Since partitions have a direct effect on performance, we would like to improve them further, perhaps by explicitly taking the size of the overlay graph into account.

Acknowledgements. We thank Ittai Abraham and Ilya Razenshteyn for their valuable input, and Christian Vetter for sharing his CH results with us.

References

1. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 230–241. Springer, Heidelberg (2011)
2. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Shortest-Path Queries in Road Networks. In: ALENEX 2007, pp. 46–59. SIAM, Philadelphia (2007)
3. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. ACM JEA 14(2.4), 1–29 (2009)

4. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. ACM JEA 15(2.3), 1–31 (2010)
5. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph Partitioning with Natural Cuts. To appear in IPDPS 2011. IEEE, Los Alamitos (2011)
6. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-Performance Multi-Level Routing. In: Demetrescu, C., et al. [9], pp. 73–92
7. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In: Lerner, J., Wagner, D., Zweig, K.A. (eds.) Algorithmics of Large and Complex Networks. LNCS, vol. 5515. Springer, Heidelberg (2009)
8. Delling, D., Wagner, D.: Pareto Paths with SHARC. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 125–136. Springer, Heidelberg (2009)
9. Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.): The Shortest Path Problem: Ninth DIMACS Implementation Challenge. DIMACS Book, vol. 74. AMS, Providence (2009)
10. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271 (1959)
11. Geisberger, R., Kobitzsch, M., Sanders, P.: Route Planning with Flexible Objective Functions. In: ALENEX 2010, pp. 124–137. SIAM, Philadelphia (2010)
12. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
13. Geisberger, R., Vetter, C.: Efficient Routing in Road Networks with Turn Costs. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 100–111. Springer, Heidelberg (2011)
14. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A^* Search Meets Graph Theory. In: SODA 2005, pp. 156–165 (2005)
15. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Reach for A*: Shortest Path Algorithms with Preprocessing. In: Demetrescu, C., et al. (eds.) [9], pp. 93–139.
16. Hilger, M., Köhler, E., Möhring, R.H., Schilling, H.: Fast Point-to-Point Shortest Path Computations with Arc-Flags. In: Demetrescu, C., et al. (eds.) [9], pp. 41–72.
17. Holzer, M., Schulz, F., Wagner, D.: Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. ACM JEA 13(2.5), 1–26 (2008)
18. Huang, Y.-W., Jing, N., Rundensteiner, E.A.: Effective Graph Clustering for Path Queries in Digital Maps. In: CIKM 1996, pp. 215–222. ACM Press, New York (1996)
19. Jung, S., Pramanik, S.: An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. IEEE TKDE 14(5), 1029–1046 (2002)
20. Karypis, G., Kumar, G.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. on Scientific Comp. 20(1), 359–392 (1999)
21. Maue, J., Sanders, P., Matijevic, D.: Goal-Directed Shortest-Path Queries Using Precomputed Cluster Distances. ACM JEA 14:3.2:1–3.2:27 (2009)
22. Muller, L.F., Zachariasen, M.: Fast and Compact Oracles for Approximate Distances in Planar Graphs. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 657–668. Springer, Heidelberg (2007)
23. Rice, M., Tsotras, V.J.: Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. In: Proc. VLDB Endowment, vol. 4(2) (2010)
24. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. ACM JEA 5(12), 1–23 (2000)
25. Schulz, F., Wagner, D., Zaroliagis, C.: Using Multi-Level Graphs for Timetable Information in Railway Systems. In: Mount, D.M., Stein, C. (eds.) ALENEX 2002. LNCS, vol. 2409. Springer, Heidelberg (2002)

Efficient Algorithms for Distributed Detection of Holes and Boundaries in Wireless Networks

Dennis Schieferdecker, Markus Völker, and Dorothea Wagner

Karlsruhe Institute of Technology (KIT), Germany
{schieferdecker,m.voelker,dorothea.wagner}@kit.edu

Abstract. We propose two novel algorithms for distributed and location-free boundary recognition in wireless sensor networks. Both approaches enable a node to decide autonomously whether it is a boundary node, based solely on connectivity information of a small neighborhood. This makes our algorithms highly applicable for dynamic networks where nodes can move or become inoperative.

We compare our algorithms qualitatively and quantitatively with several previous approaches. In extensive simulations, we consider various models and scenarios. Although our algorithms use less information than most other approaches, they produce significantly better results. They are very robust against variations in node degree and do not rely on simplified assumptions of the communication model. Moreover, they are much easier to implement on real sensor nodes than most existing approaches.

1 Introduction

Wireless sensor networks have become a prominent research topic in recent years. Their unique structure and limitations provide new and fascinating challenges. A sensor network consists of a union of small nodes that are equipped with sensing, communication, and processing capabilities. The nodes usually only have a limited view of the network. Therefore, distributed algorithms that work on local information are best suited for the emerging tasks in these environments.

Many applications in sensor networks require a certain knowledge of the underlying network topology, especially of holes and boundaries. Examples are intrusion detection, data gathering [16], mundane services like efficient routing within the network [5], or event detection [4]. In many situations, holes can also be considered as indicators for insufficient coverage or connectivity. Especially in dynamic settings, where nodes can run out of power, fail, or move, an automatic detection of holes and boundaries is inevitable.

For this reason, many boundary recognition algorithms have been developed previously. However, most of them have certain disadvantages. Some rely on oversimplified assumptions concerning the communication model or on knowledge about absolute or relative node positions, which is usually not available in large-scale sensor networks. Other algorithms are not distributed or require information exchange over long distances, so they do not scale well with network

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 388–399, 2011.
© Springer-Verlag Berlin Heidelberg 2011

size. And those algorithms that solely work locally usually produce many misclassifications. Furthermore, many of the existing algorithms are too complex for an actual implementation on real sensor nodes. So there is still demand for simple and efficient algorithms for boundary recognition.

Related Work. Since there is a wide range of applications that require boundary detection, there is an equally large number of approaches to detect holes. Based on the underlying ideas, they can be classified roughly into three categories.

Geometrical approaches use information about node positions, distances between nodes, or angular relationships. Accordingly, these approaches are limited to situations where GPS devices or similar equipment are available. Unfortunately, in many realistic scenarios this is not the case. Examples for geometrical approaches are Fang *et al.* [5], Martincic *et al.* [12], and Deogun *et al.* [3].

Statistical approaches try to recognize boundary nodes by low node degree or similar statistical properties. As long as nodes are smoothly distributed, this works quite well. However, as soon as node degrees fluctuate noticeably, most statistical approaches produce many misclassifications. Besides, these algorithms often require unrealistic high average node degrees. Prominent statistical approaches are Fekete *et al.* [6,7], and Bi *et al.* [1].

Topological approaches concentrate on information given by the connectivity graph and try to infer boundaries from its topological structure. For example, the algorithm of Kröller *et al.* [11] works by identifying complex combinatorial structures called flowers and Funke [8] and Funke *et al.* [9] describe algorithms that construct iso-contours and check whether those contours are broken. Further examples of topological approaches are given by Wang *et al.* [16], Ghrist *et al.* [10], De Silva *et al.* [2], and Saukh *et al.* [13]. A recent distributed algorithm by Dong *et al.* [4] is especially aimed at locating small holes.

An extended version of this article has been published as technical report [14]. It includes a more comprehensive list on related work and an overview on existing classification schemes for network holes and boundaries. There, we also present a more detailed description of our algorithms and additional simulation results, as we had to condense this article significantly due to page restrictions.

2 Model Description

2.1 Network Model

A sensor network consists of nodes located in the two-dimensional plane according to some distribution. Communication links between nodes induce a *connectivity graph* $\mathcal{C}(V, E)$, with graph nodes $v \in V$ corresponding to sensor nodes and graph edges $(u, v) \in E$; $u, v \in V$ to communication links between sensor nodes. An *embedding* $p : V \to \mathbb{R}^2$ of the connectivity graph $\mathcal{C}$ assigns two-dimensional coordinates $p(v)$ to each node $v \in V$. For easier reading, distances are normalized to the maximum communication distance of the sensor nodes.

Communication model. Two communication models are considered. Both assume bidirectional communication links. In the *unit disk graph* (UDG) model, two sensor nodes $u, v \in \mathcal{C}$ can communicate with each other, i.e., there exists a communication link between them, if their distance $|p(u)p(v)|$ is at most 1. In the *quasi unit disk graph* (d-QUDG) model, sensor nodes $u, v \in \mathcal{C}$ can communicate reliably if $|p(u)p(v)| \leq d$ for a given $d \in [0, 1]$. For $|p(u)p(v)| > 1$ communication is impossible. In between, communication may or may not be possible.

Node distribution. Two node distribution strategies are considered. Using *perturbed grid placement*, nodes are placed on a grid with grid spacing 0.5 and translated by a uniform random offset taken from $[0, 0.5]$ in both dimensions. Using *random placement*, nodes are placed uniformly at random on the plane.

2.2 Hole and Boundary Model

For evaluation, well-defined hole and boundary definitions are required. The definitions introduced by previous contributions are often too complicated or not extensive enough. Several approaches define holes but do not specify which nodes are considered boundary nodes, while others classify boundary nodes without taking into account their positions. In our work, we take a very practical look at what to label as holes. In short, we call large areas with no communication links crossing them holes and nodes on the borders of these areas boundary nodes.

Hole Definition. Some previous definitions are based on abstract topological definitions. In contrast, we think that the hole definition should be based on the embedding of the actual sensor network. Thus, for evaluation only, we take advantage of the true node positions.

All faces induced by the edges of the *embedded connectivity graph* $p(\mathcal{C})$ are hole candidates. Similarly to [11], we define holes to be faces of $p(\mathcal{C})$ with a circumference of at least h_{min}. Fig. 1(left) depicts a hole according to our definition. Take note that the exterior of the network is an infinite face. Thus, it is regarded as a hole for the purpose of computation and evaluation.

Boundary Node Definition. As seen in Fig. 1(left), hole borders and node locations do not have to align. Thus, there exists the problem which nodes to classify as boundary nodes. For example, it can be argued whether nodes A and B should be boundary nodes or not. To alleviate this problem, we classify nodes into three categories:

– *Mandatory Boundary Nodes.* Nodes that lie exactly on the hole border are boundary nodes.
– *Optional Boundary Nodes.* Nodes within maximum communication distance of a mandatory node can be called boundary nodes but do not have to be.
– *Interior Nodes.* All other nodes must not be classified as boundary nodes.

The resulting node classification is shown in Fig. 1(right). Mandatory boundary nodes form thin bands around holes, interrupted by structures like for nodes

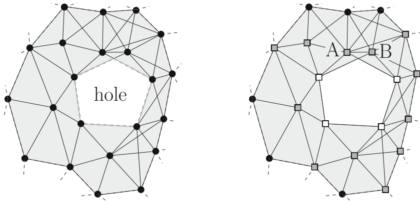


Fig. 1. (left) Hole Definition: Border as dashed line. (right) Boundary Node Classification: Mandatory nodes (white boxes), optional nodes (gray boxes), interior nodes (black circles).

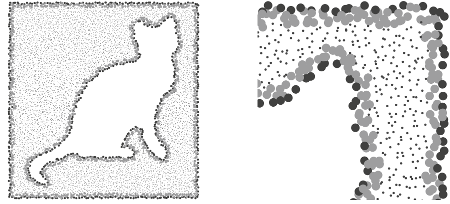


Fig. 2. Node Classification. Border outline of mandatory nodes (large, black), halo of optional nodes (large, gray), interior nodes (small, black). Full network and magnified upper right corner.

A and B before. Together with the optional boundary nodes, they form a halo around each hole. Any point within the halo is at most one maximum communication distance away from the border of the enclosed hole. A sample classification is depicted in Fig. 2.

3 Multidimensional Scaling Boundary Recognition (MDS-BR)

Both of our algorithms work in a distributed fashion and only require local connectivity information. Each node independently decides whether it is a boundary node or an interior node, solely using information from a small neighborhood.

Our first algorithm is a geometrical approach at its core. But instead of using real node coordinates, which are usually not known in sensor networks, it uses multidimensional scaling (MDS) [15] to compute virtual coordinates. Two angular conditions are then tested to classify a node, followed by a refinement step after all nodes have classified themselves. Subsequently, the base algorithm and a refinement step of MDS-BR are described. We refer to [14] for an analysis of runtime, message complexity, and possible variants of MDS-BR.

Base Algorithm. Each node performs the following base algorithm to decide independently whether to classify itself as a boundary node. At first, each node u gathers its 2-hop neighborhood N_u^2 and computes a two-dimensional embedding of $N_u^2 \cup \{u\}$, using hop distances to approximate true distances between nodes. Then, using these virtual locations, node u declares itself to be a boundary node if two conditions are fulfilled. First, the maximum opening angle α between two subsequent neighbors v, w of u in circular order and u must be larger than a threshold α_{min} as depicted in Fig. 3(a). This primary condition models the observation that boundary nodes exhibit a large gap in their neighborhood compared to interior nodes, which are usually completely surrounded by other nodes. Secondly, neighbors v, w of u must not have common neighbors

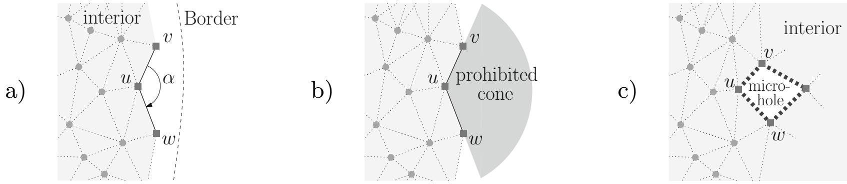


Fig. 3. MDS-BR Conditions. (a) Opening angle. (b) Prohibited cone. (c) Micro-holes.

other than u in the cone opened by (uv) and (uw). This condition is exemplified in Fig. 3(b). It filters micro-holes framed by 4 nodes with a circumference of at most 4 maximum communication distances as seen in Fig. 3(c). If such holes are to be detected, the condition can be omitted.

Both conditions only require angular information. Thus, any embedding algorithm yielding realistic angles between nodes is sufficient – we are not limited to MDS. Furthermore, we do not need complex embedding techniques to compensate for problems occurring in large graphs such as drifting or foldings since we only embed very small graphs. In particular, we only compute embeddings of 2-hop neighborhoods around each node, i.e., graphs of diameter 4 or less.

Refinement. The base algorithm already yields good results as shown in Fig. 4(a). But it retains some "noise" due to detecting boundary nodes around small holes one might not be interested in and due to some misclassifications. If desired, a refinement step can be used to remove most of these artifacts as seen in Fig. 4(b).

The refinement is performed distributed on the current set of boundary nodes. First, each boundary node u gathers its r_{min}-hop neighborhood $\tilde{N}_u^{r_{min}}$ of nodes marked as boundary nodes by the base algorithm, where r_{min} is a free parameter. Then, u verifies if there exists a shortest path of at least r_{min} hops in $\tilde{N}_u^{r_{min}} \cup \{u\}$ that contains u. If no such path exists, u classifies itself as interior node. This approach removes boundary nodes that are not part of a larger boundary structure, with r_{min} specifying the desired size of the structure. Note that only connectivity information is required for the refinement.

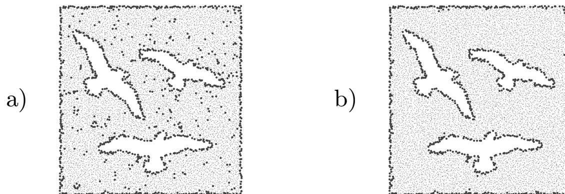


Fig. 4. Classification results of MDS-BR on a sample network. Boundary nodes marked as larger black nodes. (a) Results of the base algorithm. (b) Results after refinement.

4 Enclosing Circle Boundary Recognition (EC-BR)

Our second algorithm, EC-BR, allows to detect if a node is surrounded by other nodes without having to reconstruct node positions. Every node considers only nodes that are exactly two hops away. For a node u, we denote the corresponding subgraph as $G_u^{2\setminus 1} = (N_u^{2\setminus 1}, E_u^{2\setminus 1})$. Based on connectivity information in $G_u^{2\setminus 1}$, the node tries to decide if it is surrounded by a closed path C. If such a path exists, one can be sure that the node is not a boundary node (cf. Fig. 5(a)-(c)).

Given node positions, it would be easy to decide if an enclosing circle exists. However, we do not have this information and we do not want to reconstruct node positions in order to save computation time. So, how can we distinguish enclosing circles as the one in Fig. 5(b) and non-enclosing circles such as the one in Fig. 5(c)? Circle length is no sufficient criterion as both circles have the same length and only the first one is enclosing. Fortunately, there is a structural difference between both types of circles: in the first case, for each pair v, w of circle nodes, the shortest path between them using only circle edges is also a shortest path between them in $G_u^{2\setminus 1}$. Now we try to find a preferably long circle with this property. This can be achieved using a modified breadth-first search. The corresponding search tree for $G_u^{2\setminus 1}$ of Fig. 5(b) is depicted in Fig. 5(d). We start from a random node z in $G_u^{2\setminus 1}$ with maximum degree. In every step, we maintain shortest path lengths for all pairs of visited nodes. When a new edge is traversed, either a new node is visited or a previously encountered node is revisited. In the first case we set the shortest path distances between the already visited nodes and the new node. This can be done efficiently, as all distances can be directly inferred from the distances to the parent node. In the second case we found a new circle in $G_u^{2\setminus 1}$. The length of the circle is the current shortest path between the endpoints v and w of the traversed edge $e = (v, w)$ plus one. Subsequently, we update the shortest path information of all visited nodes. During search, we keep track of the maximum length of a circle encountered so far. Depending on this maximum length, the considered node is either classified as a boundary node or as an inner node. This enclosing circle detection can be achieved with time complexity $O(|E_u^{2\setminus 1}|)$. See [14] for further details.

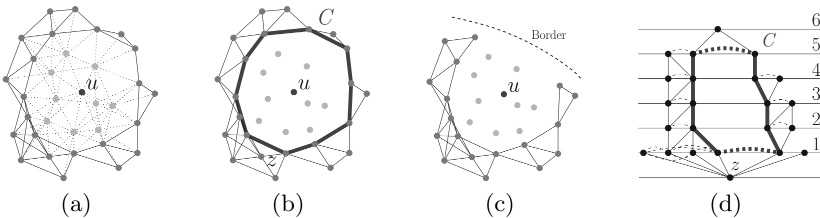


Fig. 5. Basic Idea of EC-BR. (a) 2-hop neighborhood of u. (b) Enclosing circle C. (c) Boundary node without enclosing circle. (d) Modified breadth-first search.

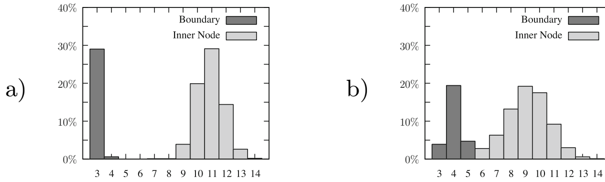


Fig. 6. Distribution of Maximum Circle Lengths. (a) UDG. (b) QUDG.

Fig. 6 depicts histograms of maximum circle lengths in our simulations with networks based on unit disk graphs and quasi unit disk graphs. There are apparently two very well defined peaks, corresponding to nodes with and without enclosing circles. Based on this distribution, we classify all nodes that have a maximum circle with length of at least 6 as inner nodes and all other nodes as boundary nodes. Our simulations indicate that this statistical classification works extremely well for both UDGs and QUDGs. Later on, we will see how good this correlates with being in the interior or on the boundary of the network.

It is also noteworthy that this kind of classification is extremely robust to variations in node degree: it does not matter whether $N_u^{2\setminus 1}$ consists of a small number of nodes or hundreds of nodes. The classification stays the same, as long as we assume that the node density is sufficiently high so that inner nodes are actually surrounded by other nodes.

Fig. 7 shows an example of a classification with EC-BR. In comparison with MDS-BR, the recognized boundaries are broader and EC-BR also detects small-scale holes which occur in areas of low node density.

Refinement. If one is only interested in large scale boundaries, EC-BR can be extended with a simple refinement. The key insight is that nodes which lie immediately next to the hole are surrounded by nodes that are marked as boundary nodes and by the hole itself. So a boundary candidate simply has to check whether a certain percentage γ of its immediate neighbors are currently classified as boundary nodes. If this is not the case, the node changes its classification to being an interior node. The effect of this simple strategy with $\gamma = 100\%$ is depicted in Fig. 7(c). Apparently, all nodes but the ones near large-scale holes are now classified as inner nodes and the boundary is very precise.

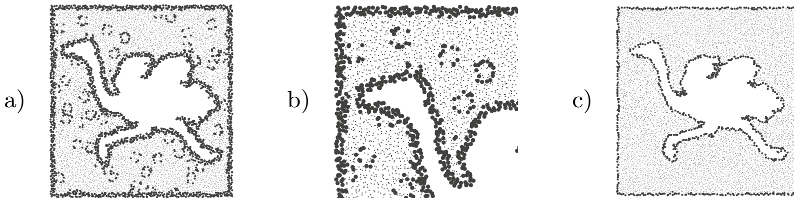


Fig. 7. Classification of EC-BR. (a,b) Before refinement. (c) After refinement.

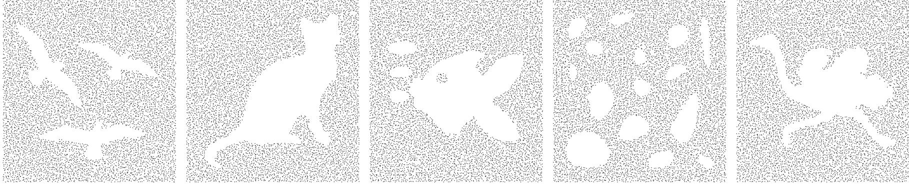


Fig. 8. Hole Patterns. Node distributions with perturbed grid placement and $d_{avg} = 12$.

5 Simulations

5.1 Simulation Setup

Network Layout. We generate network layouts by iteratively placing nodes on an area of 50×50 maximum communication distances according to one of the distribution strategies described in Section 2: perturbed grid placement (pg) or random placement (rp). After each node placement, communication links are added according to the UDG or QUDG model. Nodes are added until an average node degree d_{avg} is reached. To generate holes, we apply hole patterns such as the ones in Fig. 8. Our default layout uses perturbed grid placement, the UDG model and average node degree $d_{avg} = 12$.

Considered Algorithms. We compare the performance of our approaches EC-BR and MDS-BR to three well-known boundary recognition algorithms: The algorithm by Fekete *et al.* [7] (labelled Fekete04) and the centralized and distributed algorithms by Funke [8] and Funke *et al.* [9] (labeled Funke05 and Funke06, respectively). In addition, we show qualitative comparisons of these algorithms and the algorithm by Wang *et al.* [16] in Section 5.5. We apply our own implementation of these algorithms according to their description in the respective publications. For MDS-BR, $\alpha_{min} = 90°$ and $r_{min} = 3$ are used throughout the simulations. For the refinement of EC-BR, $\gamma = 100\%$ is used if not stated otherwise. Additional details on parameter selection are given in [14].

Measurement Procedure. Each setup is evaluated 100 times for each hole pattern in Fig. 8. Faces with circumference $h_{min} \geq 4$ are considered to be holes, e.g. a square of edge length 1 with no communication link crossing it. The analysis lists mean misclassification ratios (false negatives) in percent. For optional boundary nodes, we give the percentage of nodes classified as interior nodes.

5.2 Network Density

First, we consider the performance of all algorithms on networks with different average node degrees d_{avg}. Table 1 shows the percentage of misclassifications for mandatory boundary nodes and interior nodes with increasing d_{avg}. Results for optional boundary nodes state the percentage classified as interior nodes.

Table 1. Misclassification ratios (false negatives) in percent for average node degrees between 9 and 21 (resp. classification as interior nodes for optional boundary nodes)

	Mandatory					**Optional**					**Interior**				
	9	12	15	18	21	9	12	15	18	21	9	12	15	18	21
EC-BR	2.1	**0.0**	**0.0**	**0.0**	**0.0**	**0.3**	**0.1**	**0.2**	**0.2**	**0.3**	54.8	7.5	3.8	2.2	1.6
EC-BR Ref	4.4	0.4	0.6	1.0	1.3	51.2	80.4	81.3	82.8	84.3	**7.1**	**0.0**	**0.0**	**0.0**	**0.0**
MDS-BR	**1.9**	2.9	3.5	3.8	3.9	68.0	79.8	79.8	79.8	80.0	19.0	0.7	0.3	0.1	**0.0**
Fekete04	34.7	14.2	6.7	3.4	1.9	83.2	80.3	69.0	63.5	64.6	9.8	3.5	7.2	6.9	2.5
Funke05	16.6	6.3	5.7	5.1	5.0	61.5	59.5	55.4	52.5	50.6	21.7	3.5	2.0	1.3	0.9
Funke06	39.7	13.8	16.6	18.9	20.9	80.6	70.7	71.9	72.5	73.2	13.0	3.4	1.4	0.6	0.3

EC-BR with refinement and MDS-BR both classify almost all interior nodes correctly, except for the smallest node degree. But in this extreme scenario, all algorithms start having problems as small holes arise due to overall sparse connectivity. The classification of mandatory boundary nodes is also excellent. Here, EC-BR dominates all other algorithms. The performance of MDS-BR fluctuates only slightly over the various node degrees.

The numbers for optional boundary nodes state how many nodes within maximum communication distance of a hole are not classified as boundary nodes. Here, the results for EC-BR are particularly interesting. Before refinement, the algorithm classifies almost all of the optional nodes as boundary nodes while still providing a strict separation to the interior nodes.

A more detailed analysis shows that a lot of small-scale holes emerge in networks with average node degree of 10 or less. Accordingly, a different boundary definition and adjusted algorithms might be more appropriate for such networks. For further details we refer to our extended technical report [14].

5.3 Random Placement vs. Perturbed Grid

In this section we examine the performance when random placement is used instead of perturbed grid placement. Table 2 compares the performance of all algorithms for both placement strategies and $d_{avg} = 15$. The performance of the existing algorithms decreases dramatically compared to perturbed grid placement. In contrast, the results of the new algorithms only decrease noteworthy for interior nodes. For mandatory boundary nodes they remain roughly constant.

We also compare the influence of the network density when using random node placement. Table 3 shows the classification results for mandatory and interior nodes. Overall, our approaches dominate the other algorithms for both, mandatory boundary nodes and interior nodes. For sparse networks, we see an increased misclassification of interior nodes. This is partly caused by recognizing very small holes.

5.4 Beyond Unit Disk Graphs

Unit disk graphs are frequently used for theoretical analyses and in simulations. They are motivated by the assumption that each node has a fixed transmission range. However, under realistic conditions the transmission range depends on

Table 2. Misclassifications for random (rp) and perturbed grid placement (pg)

	Mandatory		**Interior**	
	rp	pg	rp	pg
EC-BR	**2.0**	**0.0**	48.7	3.8
EC-BR Ref	4.0	0.6	**4.1**	**0.0**
MDS-BR	5.2	3.5	12.2	0.3
Fekete04	26.2	6.7	13.0	7.2
Funke05	15.5	5.7	16.1	2.0
Funke06	45.8	16.6	7.9	1.4

Table 3. Misclassifications dependent on average node degree for random placement

	Mandatory			**Interior**		
	15	20	25	15	20	25
EC-BR	**2.0**	**1.6**	**0.5**	48.7	25.9	11.3
EC-BR Ref	4.0	3.1	1.9	**4.1**	**0.5**	**0.1**
MDS-BR	3.8	5.2	5.8	13.4	5.2	1.7
Fekete04	26.2	13.7	7.7	13.0	13.9	12.3
Funke05	15.5	9.4	6.7	16.1	8.1	3.6
Funke06	45.8	29.1	26.4	7.9	6.1	2.8

unpredictable effects such as interference or signal reflections. We now evaluate the algorithms in a more realistic context by representing uncertainties with the d-QUDG model.

Table 4 shows the performance for average node degrees 12 and 15 on 0.75-QUDG networks. The increased error rate of MDS-BR occurs because the base algorithm produces a candidate set which is not necessarily connected. Thus, the refinement classifies many correct boundary candidates as interior nodes since the connected substructures are not large enough. Fekete04, Funke05 and Funke06 yield even higher error rates and perform significantly worse than on UDGs. For QUDGs, we use $\gamma = 70\%$ for the refinement of EC-BR because in QUDGs mandatory boundary nodes are not necessarily completely surrounded by boundary candidates and holes. This value of γ works equally well for UDGs, only producing slightly broader boundaries than $\gamma = 100\%$. EC-BR with this refinement outperforms all other approaches significantly. In Table 5, we go a step further and compare 0.25-QUDGs with 0.75-QUDGs. This implies a very high level of uncertainty. As expected, all algorithms produce more misclassifications. Again, EC-BR with refinement outperforms the other algorithms easily.

5.5 Visual Comparison

We now present a visual comparison of the results of the considered algorithms in Fig. 9. Both, EC-BR with refinement and MDS-BR return thin outlines of the inner and outer border with almost no artifacts. In contrast, Funke05 returns a broader outline with more noise. The results of Fekete04 show many artifacts and a lot of boundary nodes are not detected. Funke06 correctly identifies the

Table 4. Misclassifications for the 0.75-QUDG model and different node degrees

	Mandatory		**Interior**	
	12	15	12	15
EC-BR	**0.0**	**0.0**	28.5	7.7
EC-BR Ref	**0.0**	**0.0**	**4.9**	**0.3**
MDS-BR	8.3	11.2	8.3	1.6
Fekete04	16.9	6.9	8.8	8.9
Funke05	9.0	7.4	12.9	5.2
Funke06	15.6	15.4	12.4	3.7

Table 5. Misclassifications for 0.25- and 0.75-QUDG models with node degree 12

	Mandatory		**Interior**	
	0.25	0.75	0.25	0.75
EC-BR	**3.0**	**0.0**	41.5	28.5
EC-BR Ref.	12.7	**0.0**	**1.7**	**4.9**
MDS-BR	27.7	8.3	11.8	8.3
Fekete04	14.6	16.9	13.6	8.8
Funke05	11.5	9.0	17.8	12.9
Funke06	24.2	15.6	2.8	12.4

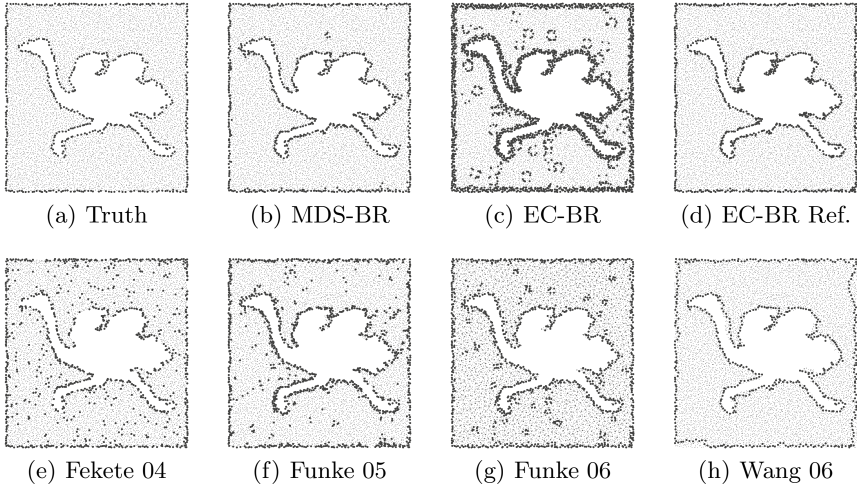


Fig. 9. Visual comparison of several algorithms for boundary detection.

boundaries with some artifacts. Similar to EC-BR, the apparent noise is caused by small holes which are surrounded by these marked nodes. We also present classification results of the global algorithm by Wang *et al.* [16]. It yields closed boundary cycles without artifacts, but due to its nature, marked boundaries are not always at the true border but shifted inwards. Hence, we did not include the algorithm into our analysis, as it would result in unfairly poor ratings.

6 Conclusion

We proposed two novel *distributed* algorithms for *location-free* boundary recognition. Both of them only depend on *connectivity information* of small *local neighborhoods*, at most 3 hops for MDS-BR and only 2 hops for EC-BR. Their *low communication overhead* makes both algorithms excellent choices for boundary recognition in large-scale sensor networks and in dynamic scenarios.

We showed in extensive simulations that both algorithms are very robust to different network densities, communication models, and node distributions. Despite their simplicity and low communication overhead, they outperformed the other considered approaches significantly. Additionally, they have much lower computational complexity than most existing approaches.

Acknowledgments

This work was supported by the German Research Foundation (DFG) within the Research Training Group GRK 1194 "Self-organizing Sensor-Actuator-Networks". We would like to thank Bastian Katz for useful discussions, as well as Yue Wang and Olga Saukh for providing us with the topologies of their respective networks.

References

1. Bi, K., Tu, K., Gu, N., Dong, W.L., Liu, X.: Topological hole detection in sensor networks with cooperative neighbors. In: International Conference on Systems and Networks Communication (ICSNC 2006), pp. 31–35 (2006)
2. De Silva, V., Ghrist, R.: Coordinate-free coverage in sensor networks with controlled boundaries via homology. International Journal of Robotics Research 25(12), 1205–1222 (2006)
3. Deogun, J.S., Das, S., Hamza, H.S., Goddard, S.: An algorithm for boundary discovery in wireless sensor networks. In: Bader, D.A., Parashar, M., Sridhar, V., Prasanna, V.K. (eds.) HiPC 2005. LNCS, vol. 3769, pp. 343–352. Springer, Heidelberg (2005)
4. Dong, D., Liu, Y., Liao, X.: Fine-grained boundary recognition in wireless ad hoc and sensor networks by topological methods. In: MobiHoc 2009: Proceedings of the Tenth ACM International Symposium on Mobile ad Hoc Networking and Computing, pp. 135–144. ACM, New York (2009)
5. Fang, Q., Gao, J., Guibas, L.J.: Locating and bypassing routing holes in sensor networks. In: INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (2004)
6. Fekete, S.P., Kaufmann, M., Kröller, A., Lehmann, N.: A new approach for boundary recognition in geometric sensor networks. In: Proceedings 17th Canadian Conference on Computational Geometry, pp. 82–85 (2005)
7. Fekete, S.P., Kröller, A., Pfisterer, D., Fischer, S., Buschmann, C.: Neighborhood-based topology recognition in sensor networks. In: Nikoletseas, S.E., Rolim, J.D.P. (eds.) ALGOSENSORS 2004. LNCS, vol. 3121, pp. 123–136. Springer, Heidelberg (2004)
8. Funke, S.: Topological hole detection in wireless sensor networks and its applications. In: DIALM-POMC 2005: Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing, pp. 44–53. ACM, USA (2005)
9. Funke, S., Klein, C.: Hole detection or: how much geometry hides in connectivity? In: SCG 2006: Proceedings of the Twenty-Second Annual Symposium on Computational Geometry, pp. 377–385. ACM, USA (2006)
10. Ghrist, R., Muhammad, A.: Coverage and hole-detection in sensor networks via homology. In: IPSN 2005: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, pp. 254–260. IEEE Press, USA (2005)
11. Kröller, A., Fekete, S.P., Pfisterer, D., Fischer, S.: Deterministic boundary recognition and topology extraction for large sensor networks. In: 17th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA 2006), pp. 1000–1009 (2006)
12. Martincic, F., Schwiebert, L.: Distributed perimeter detection in wireless sensor networks. Tech. Rep. WSU-CSC-NEWS/03-TR03, Wayne State University (2004)
13. Saukh, O., Sauter, R., Gauger, M., Marrón, P.J.: On boundary recognition without location information in wireless sensor networks. ACM Transactions on Sensor Networks (TOSN) 6(3), 1–35 (2010)
14. Schieferdecker, D., Völker, M., Wagner, D.: Efficient algorithms for distributed detection of holes and boundaries in wireless networks. Tech. Rep. 2011-08, Karlsruhe Institute of Technology (2011)
15. Torgerson, W.S.: Multidimensional Scaling: I. Theory and Method. Psychometrika 17(4), 401–419 (1952)
16. Wang, Y., Gao, J., Mitchell, J.S.: Boundary recognition in sensor networks by topological methods. In: MobiCom 2006: 12th Annual International Conference on Mobile Computing and Networking, pp. 122–133. ACM, USA (2006)

Explanations for the Cumulative Constraint: An Experimental Study*

Stefan Heinz[1] and Jens Schulz[2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
heinz@zib.de
[2] Technische Universität Berlin, Institut für Mathematik, Straße des 17. Juni 136, 10623 Berlin, Germany
jschulz@math.tu-berlin.de

Abstract. In cumulative scheduling, conflict analysis seems to be one of the key ingredients to solve such problems efficiently. Thereby, the computational complexity of explanation algorithms plays an important role. Even more when we are faced with a backtracking system where explanations need to be constructed on the fly.

In this paper we present extensive computational results to analyze the impact of explanation algorithms for the cumulative constraint in a backward checking system. The considered explanation algorithms differ in their quality and computational complexity. We present results for the domain propagation algorithms time-tabling, edge-finding, and energetic reasoning.

1 Introduction

In cumulative scheduling we are given a set of jobs that require a certain amount of different resources. In our case, the resources are renewable with a constant capacity and each job is non-interruptible with a fixed processing time and demand request for several resources. A resource can be, for example, a group of worker with the same specialization, a set of machines, or entities like power supply.

Cumulative scheduling problems have been tackled with techniques from constraint programming (CP), integer programming (IP), or satisfiability testing (SAT). In recent years hybrid approaches are developed which combine methods from these areas. Currently, the best results are reported by a hybrid solver which uses CP and SAT techniques [13]. However, there are still instances with 60 jobs and four cumulative constraints published in the PSPLib [12] that resist to be solved to proven optimality.

Several exact approaches use a search tree to solve cumulative scheduling problems. The idea is to successively divide the given problem instance into smaller subproblems until the individual subproblems are easy to solve. The best of

* Supported by the DFG Research Center Matheon *Mathematics for key technologies* in Berlin.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 400–409, 2011.
© Springer-Verlag Berlin Heidelberg 2011

all solutions found in the subproblems yields the global optimum. During the course of the algorithm, a *search tree* is created with each node representing one of the subproblems. These subproblems are usually generated by adding bound changes, called *branching decisions*, to the current problem. That means the feasibility region gets restricted. At each subproblem, mathematically sophisticated techniques exclude further values from the variables' domains. One of them is domain propagation which infers bound changes on the variables.

Recently it was discovered that *conflict analysis* plays an import role to solve cumulative scheduling problems efficiently [13,7]. Conflict analysis is used to analyze infeasible subproblems which arise during the search in order to generate *conflict clauses* [10,1] also known as *no-goods*. These conflict clauses are used to detect similar infeasible subproblems later in the search. In order to perform conflict analysis, a bound change which was performed during the search needs to be explained. Such an *explanation* is a set of bounds which infer the performed bound change. Note that bound changes which are branching decisions cannot be explained. There are two common ways to collect an explanations. One is to submit it directly with the bound change, called *forward checking*. The other way is to reconstruct an explanation on demand which means only if it is needed. This is known as *backward checking*. Both versions have their advantages and disadvantages. A forward checking system always constructs an explanation even if it is not needed, whereas the backward checking framework needs to be able to reconstruct an explanation for a bound change at any point during the search. In the latter case it can be computationally expensive to compute an explanation lazily, since the same bound changes might be explained multiple times.

In this paper we present extensive experimental results which give evidence that minimum-size explanations of bound changes are a crucial component for solving cumulative scheduling instances efficiently. We analyze the impact of constructing explanations in a backward checking system. For our study we consider the domain propagation algorithms time-tabling, edge-finding, and energetic reasoning, see [6]. In case of time-tabling, the complexity status of computing a minimum size explanation is still open. Thus, we evaluate three different heuristic approaches to deliver an appropriate explanation of small size. As benchmark set we use instances of the problem library PSPLib [12] and Pack instances from [4].

Related work. Scheduling problems have been widely studied in the literature. For an overview we refer to [4,8]. The cumulative constraint was introduced by Aggoun and Beldicneau [3]. Current state-of-the-art propagation algorithms for cumulative are surveyed in [6,4]. Best results on the instances we are focusing on are achieved by a solver combining CP and SAT techniques [13].

Learning from infeasible subproblems is one of the key ingredients in modern SAT solvers. This technique is called conflict analysis. The basic idea is to conclude a *conflict clause* which helps to prune the search tree and enables the solver to use *non-chronological backtracking*. For a detailed description see for example [10]. There are two main differences of IP and SAT solving in the context of conflict analysis. First, the variables of an IP do not need to be of binary type. Therefore, we have to extend the concept of the conflict graph: it has to

represent bound changes instead of variable fixings, see [1] for details. Second, infeasibility cannot only arise by propagation but also due to an infeasible linear program relaxation [1]. To be able to perform a conflict analysis, it is essential to explain bound changes. This means to state a set of bounds which lead to the proposed bound change. From that a conflict graph is created, then a cut in this graph is chosen, which produces a conflict clause that consists of the bound changes along the frontier of this cut. For cumulative constraints, explanation algorithms to some of the known propagation algorithms are given in [13,7].

Outline. Section 2 introduces the notation of the considered scheduling problem. In Section 3 we present the propagation and different explanation algorithms that are used in our experimental study, presented in Section 4.

2 Cumulative Scheduling

In cumulative scheduling, an instance is given by a set $\mathcal{J}$ of n non-preemptable jobs with processing times $p_j \in \mathbb{N}$ for each job $j \in \mathcal{J}$. Each job j requests a certain demand r_j of a cumulative resource with capacity $C \in \mathbb{N}$. In a contraint program, a *cumulative* constraint is given by $\texttt{cumulative}(\boldsymbol{S}, \boldsymbol{p}, \boldsymbol{r}, C)$, i.e., vectors of start times, processing times and demands, and the capacity. The cumulative constraint enforces that at each point in time t, the cumulated demand of the jobs running at t, does not exceed the given capacity, i.e.,

$$\sum_{j \in \mathcal{J} : t \in [S_j, S_j + p_j)} r_j \le C \qquad \text{for all } t.$$

Depending on the tightness of the *earliest start times* (est_j), *earliest completion times* (ect_j), *latest start times* (lst_j), and *latest completion times* (lct_j) for each job $j \in \mathcal{J}$, propagation algorithms are able to update variable bounds. Since we use start time variables S_j, the lower bound corresponds to est_j and the upper bound corresponds to lst_j.

3 Propagation and Explanation Algorithms

Explanations tend to create stronger conflict clauses if they include only few variables since we could expect that the constructed conflict graph has a smaller width and size. Hence, one would like to search for minimum sized explanations. On the other side, we are facing a backward checking system which implies that bound changes have to be explained several times during the search. Therefore, explanation algorithms should have a small complexity. In case of the cumulative constraint, computing a minimum sized explanation stands in contrast to a reasonable complexity. In this section we briefly introduce the three propagation algorithms used for our experiments. For each algorithm we state three variants to generate an explanation for a bound change. These constructions differ in their quality (the size of the explanation) and their computational complexity.

We only consider lower bound (est_j) adjustments of the start variables S_j. Upper bound (lst_j) changes can be treated symmetrically. To keep the notation simple, we assume for each interval $[a, b)$ that the condition $a < b$ holds even it is not explicitly mentioned.

3.1 Energetic Reasoning

Energetic reasoning checks non-empty time intervals $[a, b)$, with $a < b$, whether the jobs contributing to that interval require more energy than available. That is why, it has also been considered under the name *interval consistency test*. There are $O(n^2)$ intervals to be checked, see [5]. The available energy of such an interval is given by $C \cdot (b - a)$. The energy of a job is the product of its processing time and its demand. For a job j the required energy $e_j(a, b)$ for such an interval is given by:

$$e_j(a, b) := \max\{0, \min\{b - a, p_j, \text{ect}_j - a, b - \text{lst}_j\}\} \cdot r_j.$$

Hence, $e_j(a, b)$ is the non-negative minimum of (i) the energy if it runs completely in the interval $[a, b)$, i.e., $(b-a) \cdot r_j$, (ii) the energy of job j, i.e., $p_j \cdot r_j$, (iii) the left-shifted energy, i.e., $(\text{ect}_j - a) \cdot r_j$, and (iv) the right-shifted energy, i.e., $(b - \text{lst}_j) \cdot r_j$.

We can make the following deductions, see Baptiste et.al. [5] for further readings. First, in case an interval is *overloaded*, i.e., the required energy $E(a, b) := \sum_j e_j(a, b)$ is larger than $C \cdot (b - a)$, the problem is infeasible. Second, the earliest start time (lower bound) of a job j can be updated using any non-empty interval $[a, b)$ which intersects with job j according to the following equation:

$$\text{est}'_j = a + \left\lceil \frac{1}{r_j} \big(E(a, b) - e_j(a, b) - (b - a) \cdot (C - r_j) \big) \right\rceil.$$

A proof is given in [5].

In the following lemmas we state conditions on a set of bounds to achieve the deductions. In case both bounds of a job are responsible, we say, the job is reported. Otherwise, we explicitly mention the bound of interest.

Lemma 1. *An overload of interval $[a, b)$, with $a < b$, can be explained by a set $\Omega \subseteq \mathcal{J}$ such that*

$$\sum_{j \in \Omega} e_j(a, b) > C \cdot (b - a). \tag{1}$$

Lemma 2. *A lower bound update of job j to est'_j due to interval $[a, b)$, with $a < b$, intersecting with $[\text{est}_j, \text{ect}_j)$ can be explained by the previous lower bound of job j and a set $\Omega \subseteq \mathcal{J} \setminus \{j\}$ such that*

$$\sum_{i \in \Omega} e_i(a, b) > (C - r_j)(b - a) + (\text{est}'_j - a) \cdot r_j - r_j. \tag{2}$$

To construct such a sub set of jobs Ω for a lower bound update we compare in our experimental study three different algorithms:

Variant 1
Report all jobs $i \in \mathcal{J} \setminus \{j\}$ with $e_i(a,b) > 0$.
Variant 2
Report jobs $i \in \mathcal{J} \setminus \{j\}$ with $e_i(a,b) > 0$ until the Condition (2) is satisfied.
Variant 3
First, sort the jobs with respect to their energies $e_i(a,b)$ in non-increasing order and report jobs until Condition (2) is satisfied.

If the interval $[a,b)$, which inferred the lower bound change, is known, Variant 1 runs in linear time as Variant 2, which additionally needs a pre-computation of the necessary energy. Because of the sorting, Variant 3 runs in $O(n \log n)$. Observe that Variant 3 reports a minimum sized explanation with respect to interval $[a,b)$.

Note that in case of an overloaded interval (Lemma 1) the above explanation algorithms can be easily adjusted by considering the complete set of jobs $\mathcal{J}$ as basis and use Condition (1) as stopping criterion in Variants 2 and 3.

3.2 Edge-Finding

Edge-finding can be seen as a special variation of energetic reasoning. In that version the energy requirement of a job is only considered if the job lies completely in the interval $[a,b)$, i.e., $\text{est}_j \geq a$ and $\text{lct}_j \leq b$. This clearly leads to weaker bound updates, but can be executed with a smaller computational complexity using sophisticated data structures, see [14]. We use the same explanation algorithms as for energetic reasoning. Note that besides the jobs which lie completely in the interval $[a,b)$, we can also consider jobs which partly intersect with $[a,b)$. In case of constructing a suitable set of jobs Ω this has no influence on the computational complexity.

3.3 Time-Tabling

Time-tabling can be seen as a unit-interval capacity consistency test. For each interval of size one, a test is performed as by energetic reasoning. Since this attempts to be too time-consuming, implementations focus on a profile-based view. This profile is constructed using the cores of each job, see [9]. For a job j a *core* γ_j is given by the interval $[\text{lst}_j, \text{ect}_j)$. This is the interval where parts of the job j must be executed. Note that this interval might be empty.

We denote by $\text{peak}_t(\mathcal{J})$ the height of the resource profile at time t which is generated by the cores of jobs $\mathcal{J}$. Obviously, if $\text{peak}_t(\mathcal{J}) > C$ holds for some t then the corresponding cumulative constraint is infeasible. On the other hand, variable bound adjustments can be made as follows. Consider a job j. First, remove the core from job j out of the profile. Search in the interval $[\text{est}_j, \text{lst}_j]$, starting from est_j, the first time point such that job j can be scheduled without creating a profile peak exceeding the capacity. That time point est'_j defines a lower bound on the start time for job j. The bounds that are responsible in either case are stated in the following lemmas. Proofs are omitted.

Lemma 3. *An infeasibility due to* $\text{peak}_t(\mathcal{J}) > C$ *at a time point* t *can be explained by a set* $\Omega \subseteq \mathcal{J}$ *such that*

$$\sum_{j \in \Omega : t \in \gamma_j} r_j > C.$$

Lemma 4. *A lower bound update of job* j *to* est'_j *can be explained by the previous lower bound of* j *and a set* $\Omega \subseteq \mathcal{J} \setminus \{j\}$ *such that for all intervals* $I \in \{[\text{est}'_j - 1, \text{est}'_j)\} \cup \{[a, b) \subseteq [\text{est}_j, \text{est}'_j) \mid b - a = p_j\}$ *the following condition holds*

$$\exists\, t \in I : \sum_{i \in \Omega : t \in \gamma_i} r_i > C - r_j. \tag{3}$$

Note that even in the special case of jobs with unit processing times, it is an open question whether it is $\mathcal{NP}$-hard to find an explanation of minimum size. In the following we describe three different techniques to derive an explanation for a lower bound updated by the time-tabling algorithm. These approaches differ in their computational effort. Consider the lower bound update of job j from est_j to est'_j.

Variant 1
Report all variables whose core intersect with the interval $[\text{est}_j, \text{est}'_j)$.

Variant 2
(i) Sort jobs in non-decreasing order w.r.t. their demands.
(ii) For each $t \in [\text{est}_j, \text{est}'_j)$ with $\text{peak}_t(\mathcal{J} \setminus \{j\}) > C - r_j$ report jobs $i \in \mathcal{J} \setminus \{j\}$ with $t \in \gamma_i$ until Condition (3) is satisfied.

Variant 3
(i) Sort jobs in non-decreasing order w.r.t. their demands.
(ii) Set $t = \text{est}'_j - 1$.
(iii) If $t < \text{est}_j$ stop.
(iv) Explain $\text{peak}_t(\mathcal{J} \setminus \{j\})$.
(v) Find smallest time point $t' \in [t - p_j, t)$ such that $\text{peak}_{t'}(\mathcal{J} \setminus \{j\}) > C - d_j$ holds.
(vi) Set $t = t'$ and goto (iii).

Note that in Variants 2 and 3 we are starting with the largest time point, i.e., $\text{est}'_j - 1$, and report all cores until we satisfy Condition (3). For the remaining peaks we first compute the contribution of previously stated jobs and only add as many new jobs to the explanation until we fulfill Condition (3). Variant 1 runs in linear time, Variant 2 explains each peak larger than $C - d_j$, and Variant 3 tries to report only a few peaks. For the two latter once we need $O(n \log n)$ for sorting the jobs in non-decreasing order w.r.t. their demands. The number of time points that need to be considered is linear in the number of jobs.

4 Experimental Study

In this section we describe the computational environment, introduce the selected test instances, and finally present and discuss the computational results.

4.1 Computational Environment

For performing our experimental study we used the non-commercial constraint integer programming framework SCIP [2], version 2.0.1.1. We integrated CPLEX release version 12.20 as underlying linear programming solver. All computations reported were obtained using Intel Xeon 5150 core 2.66 GHz computers (in 64 bit mode) with 4 MB cache, running Linux, and 8 GB of main memory. A time limit of one hour was always enforced.

SCIP has a SAT-like conflict analysis mechanism and is a backtracking system. To avoid an overhead by constructing explanations for bound changes, it is possible to store additional information for each bound change. Since the number of stored bound changes is quite large during the search, the space for these information are restricted to 32 bits each. In case of energetic reasoning and edge finding we use these bits to store the responsible interval. Otherwise, we would need to search in worst case over $O(n^2)$ interval candidates. Hence, we can use the explanation algorithms stated in the previous section without any additional effort.

The basic version of SCIP also supports solving cumulative scheduling problem. For this study, we enhanced the capability for cumulative constraints and implemented the different explanation algorithms discussed in the previous section. We additionally used a scheduling-specific series-generation scheme based on α-points in order to generate primal solutions, see [11].

For our study we are interested in instances which are not trivial to solve on the one side and solvable on the other side. For all test sets we used the following criteria to restrict the test set to reasonable instances. We kept all instances which:

(i) could be solved to optimality by at least one solver,
(ii) at least one solver needed more than one search node, and
(iii) at least one solver needed more that one second of computational running time.

We collect resource-constrained project scheduling problem (RCPSP) instances from the problem library PSPLIB [12]. As bases we only choose test set J30 and J60, which are RCPSP instances with 30 and 60 jobs, respectively. Each test set has 480 instances. Applying the above criteria, we are left with 115 and 71 instances for the test sets J30 and J60. We omit the larger test sets J90 and J120 since for these cases the remaining set after filtering are too small. The collection of RCPSP instances in the PSPLIB is criticized for containing rather disjunctive problems. Therefore, we additionally considered the 55 Pack instances which are introduced by Artigues et.al [4]. The restricted set contains 28 instances.

4.2 Computational Results

In Section 3 we stated for the propagation algorithms time-tabling, edge-finding, and energetic reasoning three different explanation algorithms. We additionally

Table 1. Evaluation of time spend in conflict analysis for time-tabling on 115 instances from J30 and 71 instances from J60 and for energetic reasoning and edge-finding on 28 Pack instances

test set	setting	solved	outs	better	worse	totaltime	expl. time	allopt	shnodes	shtime
time-tabling										
J30	no conflict	111	4	–	–	27329.3	–	105	2267 k	6.0
	no explanation	105	10	28	29	41182.8	–	105	2477 k	6.4
	Variant 1	115	0	55	6	15739.5	0.7%	105	871 k	2.5
	Variant 2	115	0	56	4	12328.1	0.9%	105	797 k	2.5
	Variant 3	115	0	55	3	9998.9	1.02%	105	791 k	2.4
J60	no conflict	69	2	–	–	19334.3	–	60	3815 k	10.6
	no explanation	60	11	8	38	55037.5	–	60	9212 k	25.6
	Variant 1	70	1	38	10	13420.4	1.3%	60	2008 k	7.1
	Variant 2	70	1	42	5	10207.9	1.64%	60	1759 k	5.7
	Variant 3	71	0	40	5	8800.0	1.76%	60	1510 k	5.6
energetic reasoning										
Pack	no conflict	23	5	–	–	21064.0	–	16	375 k	8.2
	no explanation	21	7	9	6	29267.6	–	16	467 k	14.2
	Variant 1	21	7	3	9	30028.5	0.27%	16	641 k	18.0
	Variant 2	19	9	4	9	39323.8	0.4%	16	677 k	18.9
	Variant 3	24	4	11	3	16869.6	0.35%	16	106 k	4.3
edge-finding										
Pack	no conflict	21	7	–	–	35921.1	–	16	471 k	7.7
	no explanation	17	11	3	8	41658.3	–	16	660 k	13.1
	Variant 1	19	9	7	4	35194.8	0.018%	16	388 k	5.9
	Variant 2	19	9	6	5	36442.5	0.032%	16	378 k	5.8
	Variant 3	19	9	6	4	37720.4	0.026%	16	385 k	5.5

consider two further settings. One in which the conflict analysis is globally disabled ("no conflict"). That means no infeasible problem is analyzed. Second, the cumulative constraint does not explain the bound changes such that all bound changes made by the cumulative constraint are considered as branching decisions ("no explanation").

For each run we only used the propagation algorithm of interest for retrieving domain reductions due to the cumulative constraints. All other scheduling specific techniques were disabled. The computational results showed that the effort spent by edge-finding and energetic reasoning compared the amount of reduction detected for the RCPSP instances are rather small. Most of the running time was used within these propagation algorithms. As a result, the running time of these propagation algorithms was dominating the time needed for constructing explanation in such a way that no differences between the explanation algorithms could be made. In contrast the time-tabling algorithm is too weak for the Pack instances which ended up in similar behavior. Therefore, we only present the results of the time-tabling explanation algorithms for the rather disjunctive instances of the test set J30 and J60 and the results of the propagation algorithms edge-finding and energetic reasoning for the Pack instances.

Table 1 presents the computational results for the three test sets. Column "solved" shows how many instances were solved to proven optimality whereas

column "outs" states the number of instances which timed out. The next two columns "better" and "worse" indicate how often a setting was 10% faster or slower than the reference solver which is the one performing no conflict analysis at all. To evaluate how much time was spent by the various explanation algorithms, column "totaltime" displays the total solving time in seconds and column "expl. time" the percentage of the total solving time used for the considered explanation algorithm. The column "allopt" gives the number of instances which were solved to proven optimality by all settings. These instances are used to compute the shifted geometric mean[1] of all nodes ("shnodes") in thousands and of the running time in seconds ("shtime"), respectively.

Treating domain reductions as branching decisions ("no explanation") performs worst in all cases. The number of solved instances decreases and the shifted number of nodes and the computation times increase. This is an interesting result since overall, we experience that using conflict analysis usually helps to solve instances faster. An explanation for this behavior is that the resulting conflict clauses are large and in most case the solver discards them. This means the time spent for constructing them was in these cases useless.

In case of time-tabling we observe that Variant 3 yields the best results on instances from J30, and J60 as well. Considering only the 60 instances of the test set J60 which are solved to optimality by all settings, Variant 1 of explanation algorithms for time-tabling decreases the average running time in the shifted geometric mean by 30% and Variant 3 even to 53%, the number of nodes are decreased by 47% and 40%, respectively. Column "expl. time" reveals that the more precise the explanation algorithm is, the more percentage of the total running time is spent on explaining the bound changes. In total the time spent on explaining the cumulative propagations, is negligible. For the test set J30 the results are similar, again the strongest variant of explanations yields the best results.

In case of energetic reasoning on the highly cumulative Pack instances, we observe that only Variant 3 performs better than the "no conflict" setting. Variants 1 and 2 show that greedily explaining bound changes may mislead the search and are worse than not using conflict analysis at all. For Variant 3 the shifted solving time reduced to 50% and the shifted number of nodes decreased to 28%.

The edge-finding algorithm performs worse w.r.t. the number of solved instances when using conflict analysis in any form compared to disabling it. Two instances were not solved anymore. Recall that we explain edge-finding via the demands as defined in energetic reasoning. One could expect this to be a good counterpart, but it does not pay. There are less instances solved then by energetic reasoning. Nevertheless, in case of the instances solved to optimality by all settings, we experience that all variants need roughly the same amount of shifted nodes and shifted time which decrease by 25% and 20%, respectively, in contrast to turning conflict analysis off.

[1] The shifted geometric mean of values $t_1, \dots, t_n$ is defined as $\left(\prod(t_i + s)\right)^{1/n} - s$ with shift s. We use a shift $s = 10$ for time and $s = 100$ for nodes in order to decrease the strong influence of the very easy instances in the mean values.

5 Conclusions

We studied explanation algorithms for the cumulative constraint in a backward checking system. We presented extensive computational results. These show, that minimum sized explanations of bound changes are crucial in order to solve hard scheduling problem instances efficiently. Future research should focus on the complexity status of explanation algorithms for time-tabling or deliver approaches with reasonable computational complexity for at least some special cases.

References

1. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optimization 4(1), 4–20 (2007); special issue: Mixed Integer Programming
2. Achterberg, T.: SCIP: Solving Constraint Integer Programs. Math. Programming Computation 1(1), 1–41 (2009)
3. Aggoun, A., Beldiceanu, N.: Extending chip in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling 17(7), 57–73 (1993)
4. Artigues, C., Demassey, S., Neron, E.: Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications. ISTE (2007)
5. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-based scheduling: applying constraint programming to scheduling problems. In: International Series in Operations Research & Management Science, vol. 39. Kluwer Academic Publishers, Boston (2001)
6. Baptiste, P., Pape, C.L.: Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. Constraints 5(1/2), 119–139 (2000)
7. Berthold, T., Heinz, S., Lübbecke, M.E., Möhring, R.H., Schulz, J.: A constraint integer programming approach for resource-constrained project scheduling. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR 2010. LNCS, vol. 6140, pp. 313–317. Springer, Heidelberg (2010)
8. Hartmann, S., Briskorn, D.: A survey of variants and extensions of the resource-constrained project scheduling problem. Eur. J. Oper. Res. 207(1), 1–14 (2009)
9. Klein, R., Scholl, A.: Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. European Journal of Operational Research 112(2), 322–346 (1999)
10. Marques-Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48, 506–521 (1999)
11. Möhring, R.H., Schulz, A.S., Stork, F., Uetz, M.: Solving project scheduling problems by minimum cut computations. Manage. Sci. 49(3), 330–350 (2003)
12. PSPLib: Project Scheduling Problem LIBrary, `http://129.187.106.231/psplib/` (last accessed 2011)
13. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Explaining the cumulative propagator. Constraints, 1–33 (2010)
14. Vilím, P.: Max energy filtering algorithm for discrete cumulative resources. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 294–308. Springer, Heidelberg (2009)

GRASP with Path-Relinking for Data Clustering: A Case Study for Biological Data

Rafael M.D. Frinhani[1], Ricardo M.A. Silva[2,3], Geraldo R. Mateus[1], Paola Festa[4], and Mauricio G.C. Resende[5]

[1] Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brazil
[2] Universidade Federal de Lavras, Lavras, MG, Brazil
[3] Universidade Federal de Pernambuco, Recife, PE, Brazil
[4] University of Napoli Federico II, Napoli, Italy
[5] AT&T Labs Research, Florham Park, NJ, USA

Abstract. Cluster analysis has been applied to several domains with numerous applications. In this paper, we propose several GRASP with path-relinking heuristics for data clustering problems using as case study biological datasets. All these variants are based on the construction and local search procedures introduced by Nascimento et. al [22]. We hybridized the GRASP proposed by Nascimento et. al [22] with four alternatives for relinking method: forward, backward, mixed, and randomized. To our knowledge, GRASP with path-relinking has never been applied to cluster biological datasets. Extensive comparative experiments with other algorithms on a large set of test instances, according to different distance metrics (Euclidean, city block, cosine, and Pearson), show that the best of the proposed variants is both effective and efficient.

1 Introduction

Clustering algorithms aim to group data such that the most similar objects belong to the same group or cluster, and dissimilar objects are assigned to different clusters. According to Nascimento et. al [22], cluster analysis has been applied to several domains, natural language processing [2], galaxy formation [3], image segmentation [4], and biological data [7; 8; 9]. Surveys on clustering algorithms and their applications can be found in [5] and [6].

This paper presents a GRASP with path-relinking for data clustering based on a linearized model proposed by Nascimento et. al [22]:

$$\min \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} d_{ij} y_{ij} \tag{1}$$

subject to:

$$\sum_{k=1}^{M} x_{ik} = 1, \qquad i = 1, ..., N \tag{2}$$

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 410–420, 2011.
© Springer-Verlag Berlin Heidelberg 2011

$$\sum_{i=1}^{N} x_{ik} \geq 1, \quad k = 1, ..., M \tag{3}$$

$$x_{ik} \in \{0, 1\}, \quad i = 1, ..., N, \quad k = 1, ..., M \tag{4}$$

$$y_{ij} \geq x_{ik} + x_{jk} - 1, \quad i = 1, ..., N, \quad j = i + 1, ..., N, \quad k = 1, ..., M \tag{5}$$

$$y_{ij} \geq 0 \quad i = 1, ..., N, \quad j = i + 1, ..., N. \tag{6}$$

As described in [22], the objective function (1) aims to minimize the distance between the objects inside the same cluster, where d_{ij} denotes the distance between objects i and j; N denotes the number of objects; M denotes the number of clusters; x_{ik} is a binary variable that assumes value 1, if the object i belongs to the cluster k and 0, otherwise; and y_{ij} is a real variable that assumes the value 1, if the objects i and j belong to the same cluster.

While constraints (2) assure that object i belongs to only one cluster, constraints (3) guarantee that cluster k contains at least one object, and constraints (4) assure that the variables x_{ik} are binaries. Finally, constraints (5) and (6) guarantee that y_{ij} assumes the value 1, if both values of x_{ik} and x_{jk} are equal to 1.

The paper is organized as follows. In Section 2, we describe the GRASP with path-relinking procedure. Computational results are described in Section 3 and concluding remarks are made in Section 4.

2 GRASP with Path-Relinking for Data Clustering

GRASP, or greedy randomized adaptive search procedure, is a multi-start metaheuristic for finding approximate solutions to combinatorial optimization problems formulated as

$$\min f(x) \text{ subject to } x \in \mathcal{X},$$

where $f(\cdot)$ is an objective function to be minimized and $\mathcal{X}$ is a discrete set of feasible solutions. It was first introduced by Feo and Resende [7] in a paper describing a probabilistic heuristic for set covering. Since then, GRASP has experienced continued development [8; 23; 25] and has been applied in a wide range of problem areas [9; 10; 11].

At each GRASP iteration, a greedy randomized solution is constructed to be used as a starting solution for local search. Local search repeatedly substitutes the current solution by a better solution in the neighborhood of the current solution. If there is no better solution in the neighborhood, the current solution is declared a local minimum and the search stops. The best local minimum found over all GRASP iterations is output as the solution.

GRASP iterations are independent, i.e. solutions found in previous GRASP iterations do not influence the algorithm in the current iteration. The use of previously found solutions to influence the procedure in the current iteration can be thought of as a memory mechanism. One way to incorporate memory into GRASP is with path-relinking [13; 16]. In GRASP with path-relinking [18; 24],

an elite set of diverse good-quality solutions is maintained to be used during each GRASP iteration. After a solution is produced with greedy randomized construction and local search, that solution is combined with a randomly selected solution from the elite set using the path-relinking operator. The best of the combined solutions is a candidate for inclusion in the elite set and is added to the elite set if it meets quality and diversity criteria.

Algorithm 1 shows pseudo-code for a GRASP with path-relinking heuristic for the data clustering problem. The algorithm takes as input the dataset to be clustered and outputs the best clustering $\pi^* \in \chi$ found.

Data : Dataset to be clustered
Result : Solution $\pi^* \in \chi$.

```
1  P ← ∅;
2  while stopping criterion not satisfied do
3      π′ ← GreedyRandomized(·) as described in [22];
4      if elite set P has at least ρ elements then
5          π′ ← LocalSearch(π′) as described in [22];
6          Randomly select a solution π⁺ ∈ P;
7          π′ ← PathRelinking(π′, π⁺);
8          π′ ← LocalSearch(π′) as described in [22];
9          if elite set P is full then
10             if c(π′) ≤ max{c(π) | π ∈ P} and π′ ≉ P then
11                 Replace the element most similar to π′ among all
                      elements with cost worst than π′;
12             end
13
14         else if π′ ≉ P then
15             P ← P ∪ {π′};
16         end
17
18     else if π′ ≉ P then
19         P ← P ∪ {π′};
20     end
21 end
22 return π* = min{c(π) | π ∈ P};
```

Algorithm 1. GRASP with path-relinking heuristic

After initializing the elite set P as empty in line 1, the GRASP with path-relinking iterations are computed in lines 2 to 21 until a stopping criterion is satisfied. This criterion could be, for example, a maximum number of iterations, a target solution quality, or a maximum number of iterations without improvement. In this paper, we have adopted the maximum number of iterations without improvement (IWI) as stopping criterion of the GRASP-PR variants. During each iteration, a greedy randomized solution π' is generated in line 3. If the elite set P does not have at least ρ elements, then if π' is sufficiently different from

all other elite set solutions, π' is added to the elite set in line 19. To define the term *sufficiently different* more precisely, let $\Delta(\pi', \pi)$ be defined as the minimum number of moves needed to transform π' into π or vice-versa. For a given level of difference δ, we say that π' is sufficiently different from all elite solutions in P if $\Delta(\pi', \pi) > \delta$ for all $\pi \in P$, which we indicate with the notation $\pi' \not\approx P$. If the elite set P does have at least ρ elements, then the steps in lines 5 to 16 are computed.

The local search described in [22] is applied in line 5 using π' as a starting point, resulting in a local minimum, which we denote by π'. Next, path-relinking is applied in line 7 between π' and an elite solution π^+, randomly chosen in line 6. Solution π^+ is selected with probability proportional to $\Delta(\pi', \pi^+)$. In line 8, the local search described in [22] is applied to π'. If the elite set is full, then if π' is of better quality than the worst elite solution and $\pi' \not\approx P$, then it will be added to the elite set in line 11 in place of some elite solution. Among all elite solutions having cost no better than that of π', a solution π most similar to π', i.e. with the smallest $\Delta(\pi', \pi)$ value, is selected to be removed from the elite set. Ties are broken at random. Otherwise, if the elite set is not full, π' is simply added to the elite set in line 15 if $\pi' \not\approx P$.

2.1 Path-Relinking

Path-relinking was originally proposed by Glover [13] as an intensification strategy exploring trajectories connecting elite solutions obtained by tabu search or scatter search [14; 15; 16]. Starting from one or more elite solutions, paths in the solution space leading toward other elite solutions are generated and explored in the search for better solutions. To generate paths, moves are selected to introduce attributes in the current solution that are present in the elite guiding solution. Path-relinking may be viewed as a strategy that seeks to incorporate attributes of high quality solutions, by favoring these attributes in the selected moves.

Algorithm 2 illustrates the pseudo-code of the path-relinking procedure applied to a pair of solutions x_s (starting solution) and x_t (target solution). The procedure starts by computing the symmetric difference $\Delta(x_s, x_t)$ between the two solutions, i.e. the set of moves needed to reach x_t (target solution) from x_s (initial solution). A path of solutions is generated linking x_s and x_t. The best solution x^* in this path is returned by the algorithm. At each step, the procedure examines all moves $m \in \Delta(x, x_t)$ from the current solution x and selects the one which results in the least cost solution, i.e. the one which minimizes $f(x \oplus m)$, where $x \oplus m$ is the solution resulting from applying move m to solution x. The best move m^* is made, producing solution $x \oplus m^*$. The set of available moves is updated. If necessary, the best solution x^* is updated. The procedure terminates when x_t is reached, i.e. when $\Delta(x, x_t) = \emptyset$.

We notice that path-relinking may also be viewed as a constrained local search strategy applied to the initial solution x_s, in which only a limited set of moves can be performed and where uphill moves are allowed. Several alternatives have

Data : Starting solution x_s and target solution x_t
Result : Best solution x^* in path from x_s to x_t
Compute symmetric difference $\Delta(x_s, x_t)$;
$f^* \leftarrow \min\{f(x_s), f(x_t)\}$;
$x^* \leftarrow \text{argmin}\{f(x_s), f(x_t)\}$;
$x \leftarrow x_s$;
while $\Delta(x, x_t) \neq \emptyset$ **do**
    $m^* \leftarrow \text{argmin}\{f(x \oplus m) : m \in \Delta(x, x_t)\}$;
    $\Delta(x \oplus m^*, x_t) \leftarrow \Delta(x, x_t) \setminus \{m^*\}$;
    $x \leftarrow x \oplus m^*$;
    if $f(x) < f^*$ **then**
        $f^* \leftarrow f(x)$;
        $x^* \leftarrow x$;
    end
end

Algorithm 2. Path-relinking

been considered and combined in recent implementations of path-relinking [1; 2; 3; 5; 26; 27; 29], among them:

- *forward relinking:* path-relinking is applied using the worst among x_s and x_t as the initial solution and the other as the target solution;
- *backward relinking:* the roles of x_s and x_t are interchanged, path-relinking is applied using the best among x_s and x_t as the initial solution and the other as the target solution;
- *mixed relinking:* two paths are simultaneously explored, the first emanating from x_s and the second from x_t, until they meet at an intermediary solution equidistant from x_s and x_t; and
- *randomized relinking:* instead of selecting the best yet unselected move, randomly select one from among a candidate list with the most promising moves in the path being investigated.

Figure 3 illustrates an example of path-relinking. Let x be a solution composed by clusters $A = \{2, 3, 7\}$, $B = \{4, 6\}$, and $C = \{1, 5\}$; and x_t the target solution with the clusters $A = \{6, 7\}$, $B = \{4, 5\}$, and $C = \{1, 2, 3\}$. Initially, $\Delta(x, x_t) = \{(2, A, C), (3, A, C), (5, C, B), (6, B, A)\}$, where (e, s, t) means a move of element e from cluster s to cluster t. After the best move $(2, A, C)$ from solution x is performed, x is updated with clusters $A = \{3, 7\}$, $B = \{4, 6\}$, and $C = \{1, 2, 5\}$. The process is repeated until x_t is reached.

3 Experimental Results

In this section, we present results on computational experiments with the GRASP with path-relinking (GRASP-PR) heuristic introduced in this paper. First, we describe our datasets. Second, we describe our test environment and determine an appropriated combination of values for the parameters of the heuristic. Finally,

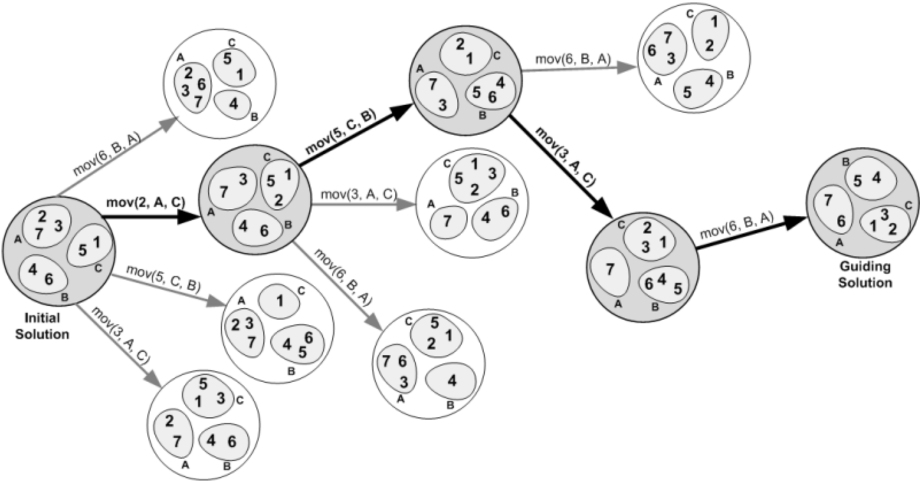


Fig. 1. A path-relinking example for data clustering

besides the GRASP-L algorithm introduced by Nascimento [22], we compare several GRASP-PR variants implementations with the three known clustering algorithms described in [22]: K-means, K-medians and PAM [17][1].

3.1 Datasets

We used the same five datasets from [22]: fold protein classification, named Protein [6], prediction of protein localization sites, named Yeast [21]; seven cancer diagnosis datasets, named Breast [4], Novartis [30], BreastA [31], BreastB [32], DLBCLA [20], DLBCLB [28] and MultiA [30]; and the benchmark Iris [12].

Table 1. Characteristics of datasets used in the experiments

Data Set	#Objects	#Str(#Groups)	#Attrib
Protein	698	2 (4,27)	125
Yeast	1484	1 (10)	8
Breast	699	2 (2,8)	9
Novartis	103	1 (4)	1000
BreastA	98	1 (3)	1213
BreastB	49	2 (2,4)	1213
DLBCLA	141	1 (3)	661
DLBCLB	180	1 (3)	661
MultiA	103	1 (4)	5565
Iris	140	1 (3)	4

[1] K-means and K-medians implementations are available at http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/cluster/software.htm

Table 1 shows the main characteristics of each dataset. The second column indicates the number of objects in each dataset. The third column shows the number of structures in the dataset and, in parenthesis, the number of clusters for each structure. The fourth column shows the number of attributes in the objects. Next, we describe in more details each of the datasets used.

3.2 Test Environment and Parameters for GRASP-PR Heuristic

All experiments with GRASP-PR were done on a Dell computer with Core 2 Duo 2.1 GHz T8100 Intel processor and 3 Gb of memory, running Windows XP Professional version 5.1 2002 SP3 x86. The GRASP-PR heuristic was implemented in Java and compiled into bytecode with `javac` version 1.6.0.20. The random-number generator is an implementation of the Mersenne Twister algorithm [19] from the COLT[2] library.

The values of the parameters for GRASP-PR heuristic used for each dataset are shown in Table 2.

Table 2. Path-Relinking parameters. Pool size (PS), elements in pool before start PR (EPBS), symmetrical difference (SD), and Iterations without Improvement (IWI)

	Iris	Novartis	BrstA	BrstB1	BrstB2	DLBCLA	DLBCLB	MultA	Brst1	Brst2	Prt1	Prt2	Yeast
PS	3	5	4	3	3	5	5	5	3	6	5	5	7
EPBS	1	3	1	1	1	2	2	2	1	3	2	3	3
SD	4	70	4	30	30	100	100	70	4	550	450	450	1200
IWI	15	15	15	15	15	15	15	15	15	15	15	15	5

3.3 Numerical Comparisons

We compare the three known clustering algorithms described in [22] (K-means, K-medians and PAM [17]) with the GRASP-L algorithm introduced by Nascimento [22] and the following five GRASP-PR variants implementations: GRASP, GRASP-PRf, GRASP-PRb, GRASP-PRm and GRASP-PRrnd. GRASP is our implementation of the GRASP-L algorithm. GRASP-PRf, GRASP-PRb, GRASP-PRm and GRASP-PRrnd correspond to the following relinking alternatives: forward, backward, mixed and greedy randomized, respectively. We used the same distance measurements for all of them.

The comparisons of the algorithms were based on the Corrected Rand index (CRand) proposed in [26] (Table 3). While GRASP-L, K-means and K-medians were run 100 times, GRASP-PRf, GRASP-PRb, GRASP-PRm and GRASP-PRrnd were run 30 times. All algorithms selected the partition with the best solution for each of the distance metrics.

With respect to the comparisons of the algorithms based on the Corrected Rand index (CRand) reported in Table 3, we observe that GRASP-PR variants found the best-quality solutions with all different dissimilarity measures. In fact,

[2] COLT is a open source library for high performance scientific and technical computing in Java. See `http://acs.lbl.gov/~hoschek/colt/`

Table 3. Summary of CRand results for GRASP-PRrnd, GRASP-PRm, GRASP-PRb, GRASP-PRf, GRASP, GRASP-L, K-means, K-medians and PAM algorithms. M is the number of clusters for the best CRand found. Times are given in seconds on a Core 2 Duo 2.1 GHz T8100 Intel processor (`javac` compiler version 1.6.0.20). Times for GRASP-L and PAM algorithms are not reported in [22].

	GRASP-PRrnd			GRASP-PRm			GRASP-PRb			GRASP-PRf			GRASP			GRASP-L		KMEANS			KMEDIANS			PAM	
	M	cRand	Time	M	cRand	Time	M	cRand	Time	M	cRand	Time	M	cRand	Time	M	cRand	M	cRand	Time	M	cRand	Time	M	cRand
EUCLIDEAN																									
Protein	4	0.297	58.906	4	0.297	63.624	4	0.294	58.625	4	0.294	61.672	4	0.294	54.531	4	**0.322**	7	**0.322**	0.484	7	0.313	0.453	6	0.250
	11	**0.169**	207.890	11	0.168	306.197	11	0.167	244.547	11	**0.169**	156.719	11	0.168	122.094	11	0.168	17	0.139	0.562	25	0.134	0.625	13	0.098
Breast	2	**0.878**	16.531	2	**0.878**	19.781	2	**0.878**	19.297	2	**0.878**	19.343	2	**0.878**	18.484	2	0.877	2	0.803	0.078	2	0.782	0.062	2	0.828
	15	0.016	147.282	15	0.016	137.857	15	0.014	129.203	15	0.016	152.203	15	0.017	124.625	15	0.015	18	-0.010	0.094	17	**0.036**	0.078	5	0.012
Yeast	9	0.151	1689.766	9	0.153	1410.047	9	0.153	1492.132	9	0.150	849.363	9	0.151	1738.641	9	0.150	7	0.170	0.109	8	**0.173**	0.141	8	0.143
Novartis	4	**0.950**	7.391	4	**0.950**	6.921	4	**0.950**	6.984	4	**0.950**	7.344	4	**0.950**	6.094	4	0.921	4	0.946	0.469	4	0.946	0.484	4	0.897
BreastA	2	0.682	5.937	2	**0.723**	5.844	2	0.682	5.797	2	0.682	6.188	2	0.682	5.813	2	0.682	2	0.654	0.468	2	0.654	0.484	2	0.543
BreastB	2	**0.694**	1.890	2	**0.694**	1.906	2	**0.694**	1.906	2	**0.694**	1.985	2	**0.694**	1.891	2	0.626	3	0.502	0.281	4	0.500	0.297	2	0.388
	2	**0.322**	1.890	2	**0.322**	2.031	2	**0.322**	1.890	2	**0.322**	1.984	2	0.321	1.891	2	0.314	3	0.286	0.297	3	0.260	0.281	2	0.187
DLBCLA	4	0.465	13.640	4	0.431	10.187	4	**0.504**	7.015	4	0.408	8.297	4	0.443	7.828	4	0.408	4	0.309	0.437	5	0.365	0.437	4	0.276
DLBCLB	4	0.520	13.578	4	0.519	21.661	4	**0.537**	11.000	4	0.509	13.219	4	0.519	10.391	4	0.481	2	0.420	0.515	3	0.424	0.515	3	0.391
MultiA	4	**0.874**	34.031	4	**0.874**	31.562	4	**0.874**	31.781	4	**0.874**	33.859	4	**0.874**	29.844	4	0.874	6	0.765	2.500	5	0.682	2.500	4	0.765
Iris	3	**0.757**	0.312	3	**0.757**	0.312	3	**0.757**	0.390	3	**0.757**	0.391	3	**0.757**	0.359	3	0.756	3	0.730	0.046	3	0.744	0.063	3	0.730
CITY BLOCK																									
Protein	5	**0.310**	71.141	5	**0.310**	85.748	5	0.309	62.484	5	0.309	50.812	5	0.309	98.359	5	0.293	8	0.223	0.484	7	0.229	0.468	3	0.192
	9	0.179	95.187	9	0.176	254.432	9	**0.180**	76.078	9	**0.180**	115.859	9	0.175	99.032	9	0.166	17	0.158	0.547	28	0.141	0.594	19	0.084
Breast	2	**0.877**	14.562	2	**0.877**	17.672	2	**0.877**	17.344	2	**0.877**	17.234	2	**0.877**	16.422	2	**0.877**	2	0.770	0.078	2	0.765	0.063	2	0.807
	19	0.016	243.656	19	0.015	295.462	19	0.015	166.110	19	0.015	210.172	19	0.016	146.281	19	0.013	19	-0.009	0.094	10	**0.023**	0.063	13	0.010
Yeast	7	0.161	1432.047	7	0.159	953.766	7	0.160	1374.019	7	0.161	1630.917	7	0.161	706.266	7	0.157	7	**0.181**	0.109	6	0.167	0.110	7	0.152
Novartis	4	**0.950**	2.859	4	**0.950**	2.796	4	**0.950**	2.781	4	**0.950**	2.796	4	**0.950**	2.500	4	0.921	4	0.946	0.484	4	0.921	0.484	4	0.947
BreastA	2	**0.723**	1.844	2	**0.723**	1.889	2	0.682	1.797	2	**0.723**	1.922	2	0.722	1.750	2	0.682	2	0.583	0.484	2	0.618	0.484	4	0.560
BreastB	4	0.366	1.391	4	0.366	1.343	4	0.261	0.922	4	0.288	2.250	4	0.328	0.890	4	0.228	3	**0.563**	0.297	2	0.561	0.297	2	0.388
	7	0.333	2.297	7	0.344	1.828	7	0.244	0.953	7	0.328	1.265	7	**0.367**	1.000	7	0.159	3	0.328	0.297	3	0.284	0.312	2	0.187
DLBCLA	3	**0.838**	1.953	3	**0.838**	1.999	3	**0.838**	1.953	3	**0.838**	1.954	3	**0.838**	1.797	3	0.800	3	0.805	0.422	3	0.784	0.453	3	0.406
DLBCLB	2	0.701	2.906	2	0.701	2.797	2	**0.703**	2.860	2	0.701	2.843	2	0.701	2.672	2	0.700	2	0.690	0.500	2	0.690	0.500	3	0.350
MultiA	4	0.899	10.297	4	**0.924**	11.141	4	0.899	10.438	4	0.899	11.015	4	0.899	9.906	4	0.899	4	0.851	2.300	4	0.875	2.300	5	0.750
Iris	3	**0.818**	0.281	3	**0.818**	0.281	3	**0.818**	0.343	3	**0.818**	0.359	3	**0.818**	0.328	3	**0.818**	3	0.717	0.063	3	0.717	0.047	3	0.772
COSINE																									
Protein	4	**0.350**	102.668	4	0.348	89.419	4	0.344	168.344	4	0.342	81.656	4	0.348	82.296	4	0.349	7	0.320	0.438	6	0.304	0.469	6	0.247
	12	0.170	266.781	12	0.170	141.794	12	0.170	162.500	12	**0.173**	225.656	12	0.170	150.578	12	0.166	20	0.134	0.656	21	0.125	0.625	15	0.091
Breast	3	0.294	28.781	3	0.294	32.812	3	0.294	29.547	3	0.294	31.796	3	0.294	31.188	3	0.293	4	0.258	0.078	3	0.306	0.078	3	**0.332**
	8	0.022	133.453	8	0.021	77.403	8	0.021	78.281	8	0.022	82.703	8	0.022	77.562	8	0.020	2	0.027	0.078	8	**0.052**	0.078	3	0.021
Yeast	9	0.137	1103.942	9	0.137	972.313	9	0.137	680.172	9	0.137	988.547	9	0.136	716.172	9	0.135	9	0.138	0.156	6	0.132	0.125	7	**0.146**
Novartis	4	**0.950**	14.422	4	**0.950**	12.328	4	**0.950**	13.578	4	**0.950**	11.734	4	**0.950**	11.766	4	0.920	4	0.919	0.484	4	0.919	0.484	4	0.745
BreastA	2	0.687	12.282	2	0.687	10.996	2	0.687	12.172	2	0.687	10.485	2	0.687	12.281	2	0.686	2	**0.691**	0.485	2	**0.691**	0.484	2	0.664
BreastB	2	**0.694**	3.016	2	**0.694**	2.875	2	**0.694**	2.891	2	**0.694**	2.688	2	**0.694**	3.015	2	0.626	2	0.561	0.281	3	0.502	0.281	4	0.443
	2	**0.322**	3.000	2	**0.322**	2.875	2	**0.322**	3.031	2	**0.322**	2.687	2	0.321	3.016	2	0.314	2	0.269	0.297	3	0.264	0.281	4	0.239
DLBCLA	4	0.607	13.203	4	0.619	16.640	4	0.607	12.422	4	0.607	12.078	4	0.607	11.297	4	0.605	5	0.642	0.468	4	**0.678**	0.438	3	0.547
DLBCLB	4	0.500	24.375	4	0.500	21.109	4	0.500	23.281	4	0.500	19.984	4	0.500	21.797	4	0.502	3	0.501	0.516	3	**0.623**	0.562	5	0.385
MultiA	4	**0.831**	60.078	4	**0.831**	54.468	4	**0.831**	56.140	4	**0.831**	48.532	4	**0.831**	47.875	4	0.805	4	0.718	2.300	7	0.731	2.400	6	0.716
Iris	3	**0.942**	0.359	3	**0.942**	0.344	3	**0.942**	0.421	3	**0.942**	0.406	3	**0.942**	0.390	3	0.941	3	0.904	0.047	3	0.941	0.062	3	0.904
PEARSON																									
Protein	4	**0.345**	133.500	4	**0.345**	127.592	4	**0.345**	135.563	4	**0.345**	139.625	4	**0.345**	120.532	4	0.344	7	0.313	0.500	7	0.306	0.484	6	0.245
	12	**0.172**	322.891	12	**0.172**	251.089	12	**0.172**	328.969	12	0.169	261.187	12	**0.172**	437.438	12	0.167	20	0.129	0.609	27	0.136	0.641	14	0.096
Breast	3	0.311	38.515	3	0.311	41.327	3	0.311	40.968	3	0.311	42.141	3	0.311	38.703	3	0.284	2	**0.441**	0.078	2	0.368	0.078	2	0.289
	11	0.017	137.078	11	0.016	197.056	11	0.017	124.360	11	0.016	110.844	11	0.016	91.234	11	0.017	9	0.015	0.093	19	**0.024**	0.109	6	0.015
Yeast	9	0.138	1010.289	9	0.138	877.000	9	0.138	662.390	9	0.138	660.406	9	0.138	510.969	9	0.131	8	0.135	0.171	8	0.133	0.125	7	**0.145**
Novartis	4	**0.950**	23.625	4	**0.950**	20.156	4	**0.950**	22.016	4	**0.950**	23.422	4	**0.950**	18.110	4	0.920	4	0.919	0.468	4	0.919	0.484	4	0.746
BreastA	2	0.692	20.656	2	0.692	18.904	2	0.692	20.547	2	0.692	21.734	2	0.692	20.625	2	0.692	2	**0.705**	0.469	2	**0.705**	0.469	2	0.635
BreastB	2	**0.766**	4.797	2	**0.766**	5.562	2	0.626	4.719	2	**0.766**	5.546	2	**0.766**	6.546	2	0.694	3	0.502	0.297	3	0.529	0.297	3	0.445
	2	0.322	8.969	2	0.322	5.016	2	0.306	4.734	2	0.281	5.563	2	0.279	6.468	2	**0.355**	4	0.289	0.296	3	0.283	0.281	3	0.227
DLBCLA	4	0.604	20.688	4	0.604	20.577	4	0.607	20.844	4	0.604	23.140	4	0.604	18.110	4	0.585	4	0.605	0.469	4	**0.684**	0.453	4	0.586
DLBCLB	2	0.585	32.737	2	0.585	33.796	2	0.585	36.937	2	0.585	39.641	2	0.585	37.453	2	0.527	3	**0.665**	0.547	3	0.561	0.516	3	0.545
MultiA	4	**0.829**	98.766	4	**0.829**	92.655	4	**0.829**	92.453	4	**0.829**	102.156	4	**0.829**	85.719	4	0.828	4	0.718	2.300	9	0.691	2.500	4	0.705
Iris	3	0.886	0.750	3	0.886	0.640	3	0.886	0.781	3	0.886	0.781	3	0.886	0.766	3	0.886	3	0.886	0.047	3	**0.941**	0.062	3	0.886

- using Euclidean metric as dissimilarity measure, GRASP-PRb found best results for 8 out of 10 datasets; GRASP-PRrnd, GRASP-PRm and GRASP-PRf found best results for 7 datasets; GRASP for 5, GRASP-L for 1, while K-means and K-medians found the best solution for only 1 and 2 datasets, respectively;
- using City Block metric as dissimilarity measure, GRASP-PRm found best results for 7 out of 10 datasets; GRASP-PRrnd, GRASP-PRb and GRASP-PRf found best results for 6 datasets; GRASP for 5, GRASP-L and K-medians for 2, while K-means only for 1;
- using Cosine metric as dissimilarity measure, GRASP-PRrnd and GRASP-PRf found best results for 6 out of 10 datasets; GRASP-PRb and GRASP-PRm for 5, GRASP for 4, and K-medians, PAM, and K-means for 4, 2, and 1, respectively.
- using Pearson metric, GRASP-PRrnd, GRASP-PRm and GRASP found best results for 5 out of 10 datasets; GRASP-PRb, GRASP-PRf and K-medians for 4, K-means for 3, GRASP-L and PAM for 1.

As expected, we can observe in In Table 3 that the higher the distance metric complexity, the higher the computacional time. In instances with a small number of clusters, usually the GRASP procedure is sufficient to find good solutions, reducing the path-relinking utility and increasing the time consuming. The experimental results present the number of clusters with the highest CRand in the range from 2 to 30 clusters.

4 Concluding Remarks

In this paper, we propose four variants of GRASP with path-relinking (forward, backward, mixed, and randomized) for data clustering problem. The algorithms were implemented in Java and extensively tested. Computational results from several instances from the literature demonstrate that the heuristic is a well-suited approach for data clustering.

Acknowledgment

The research of R.M.A. Silva was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq), the Foundation for Support of Research of the State of Minas Gerais, Brazil (FAPEMIG), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brazil (CAPES), and Fundação de Apoio ao Desenvolvimento da UFPE, Brazil (FADE).

References

[1] Aiex, R.: Uma investigação experimental da distribuição de probabilidade de tempo de solução em heurísticas GRASP e sua aplicação na análise de implementações paralelas. Ph.D. thesis, Department of Computer Science, Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil (2002)

[2] Aiex, R., Binato, S., Resende, M.: Parallel GRASP with path-relinking for job shop scheduling. Parallel Computing 29, 393–430 (2003)

[3] Aiex, R., Resende, M., Pardalos, P., Toraldo, G.: GRASP with path relinking for the three-index assignment problem. INFORMS J. on Computing 17(2), 224–247 (2005)

[4] Bennett, K.P., Mangasarian, O.: Robust linear programming discrimination of two linearly inseparable sets. Optimization Methods and Software 1(1), 23–34 (1992)

[5] Binato, S., Faria Jr., H., Resende, M.: Greedy randomized adaptive path relinking. In: Sousa, J. (ed.) Proceedings of the IV Metaheuristics International Conference, pp. 393–397 (2001)

[6] Ding, C., Dubchak, I.: Multi-class protein fold recognition using support vector machines and neural networks. Bioinformatics 17(4), 349–358 (2001)

[7] Feo, T., Resende, M.: A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters 8, 67–71 (1989)

[8] Feo, T., Resende, M.: Greedy randomized adaptive search procedures. J. of Global Optimization 6, 109–133 (1995)

[9] Festa, P., Resende, M.: GRASP: An annotated bibliography. In: Ribeiro, C., Hansen, P. (eds.) Essays and Surveys on Metaheuristics, pp. 325–367. Kluwer Academic Publishers, Dordrecht (2002)

[10] Festa, P., Resende, M.: An annotated bibliography of GRASP – Part I: Algorithms. International Transactions on Operational Research 16, 1–24 (2009)

[11] Festa, P., Resende, M.: An annotated bibliography of GRASP – Part II: Applications. International Transactions on Operational Research (2009)

[12] Fisher, R., et al.: The use of multiple measurements in taxonomic problems. Annals of Eugenics 7, 179–188 (1936)

[13] Glover, F.: Tabu search and adaptive memory programing – Advances, applications and challenges. In: Barr, R., Helgason, R., Kennington, J. (eds.) Interfaces in Computer Science and Operations Research, pp. 1–75. Kluwer, Dordrecht (1996)

[14] Glover, F.: Multi-start and strategic oscillation methods – Principles to exploit adaptive memory. In: Laguna, M., Gonzáles-Velarde, J. (eds.) Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research, pp. 1–24. Kluwer, Dordrecht (2000)

[15] Glover, F., Laguna, M.: Tabu Search. Kluwer, Dordrecht (1997)

[16] Glover, F., Laguna, M., Martí, R.: Fundamentals of scatter search and path relinking. Control and Cybernetics 39, 653–684 (2000)

[17] Kaufman, L., Rousseeuw, P.: Finding groups in data: an introduction to cluster analysis. WileyBlackwell (2005)

[18] Laguna, M., Martí, R.: GRASP and path relinking for 2-layer straight line crossing minimization. INFORMS Journal on Computing 11, 44–52 (1999)

[19] Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation 8, 3–30 (1998)

[20] Monti, S., Savage, K., Kutok, J., Feuerhake, F., Kurtin, P., Mihm, M., Wu, B., Pasqualucci, L., Neuberg, D., Aguiar, R., et al.: Molecular profiling of diffuse large B-cell lymphoma identifies robust subtypes including one characterized by host inflammatory response. Blood 105(5), 1851–1861 (2005)

[21] Nakai, K., Kanehisa, M.: Expert system for predicting protein localization sites in gram-negative bacteria. Proteins: Structure, Function, and Bioinformatics 11(2), 95–110 (1991)

[22] Nascimento, M., Toledo, F., de Carvalho, A.: Investigation of a new GRASP-based clustering algorithm applied to biological data. Computers & Operations Research 37(8), 1381–1388 (2010)
[23] Resende, M., Ribeiro, C.: Greedy randomized adaptive search procedures. In: Glover, F., Kochenberger, G. (eds.) Handbook of Metaheuristics, pp. 219–249. Kluwer Academic Publishers, Dordrecht (2002)
[24] Resende, M., Ribeiro, C.: GRASP with path-relinking: Recent advances and applications. In: Ibaraki, T., Nonobe, K., Yagiura, M. (eds.) Metaheuristics: Progress as Real Problem Solvers, pp. 29–63. Springer, Heidelberg (2005)
[25] Resende, M., Ribeiro, C.: Greedy randomized adaptive search procedures: Advances and applications. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics, 2nd edn., pp. 281–317. Springer Science+Business Media (2010)
[26] Ribeiro, C., Rosseti, I.: Efficient Graph Rewriting and Its Implementation. LNCS 2004, pp. 922–926 (2002)
[27] Ribeiro, C., Uchoa, E., Werneck, R.: A hybrid GRASP with perturbations for the Steiner problem in graphs. INFORMS Journal on Computing 14, 228–246 (2002)
[28] Rosenwald, A., Wright, G., Chan, W., Connors, J., Campo, E., Fisher, R., Gascoyne, R., Muller-Hermelink, H., Smeland, E., Staudt, L.: The use of molecular profiling to predict survival after chemotherapy for diffuse Large-B-cell lymphoma. The New England Journal of Medicine 346(25), 1937–1947 (2002)
[29] Rosseti, I.: Heurísticas para o problema de síntese de redes a 2-caminhos. Ph.D. thesis, Department of Computer Science, Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil (July 2003)
[30] Su, A., Cooke, M., Ching, K., Hakak, Y., Walker, J., Wiltshire, T., Orth, A., Vega, R., Sapinoso, L., Moqrich, A., et al.: Large-scale analysis of the human and mouse transcriptomes. Proceedings of the National Academy of Sciences 99(7), 4465–4470 (2002)
[31] Van't, V., Laura, J., Hongyue, D., Vijver, M.V.D., He, Y., Hart, A., et al.: Gene expression profiling predicts clinical outcome of breast cancer. Nature 415(6871), 530–536 (2002)
[32] West, M., Blanchette, C., Dressman, H., Huang, E., Ishida, S., Spang, R., Zuzan, H., Olson, J., Marks, J., Nevins, J.: Predicting the clinical status of human breast cancer by using gene expression profiles. Proceedings of the National Academy of Sciences of the United States of America 98(20), 11462 (2001)

An Iterative Refinement Algorithm for the Minimum Branch Vertices Problem

Diego M. Silva[1,2], Ricardo M.A. Silva[1,3], Geraldo R. Mateus[2], José F. Gonçalves[4], Mauricio G.C. Resende[5], and Paola Festa[6]

[1] Dept. of Computer Science, Federal University of Lavras
C.P. 3037, CEP 37200-000, Lavras, MG, Brazil
diego.silva@gmail.com, rmas@dcc.ufla.br

[2] Dept. of Computer Science, Federal University of Minas Gerais
C.P. 702, CEP 31270-010, Belo Horizonte, MG, Brazil
mateus@dcc.ufmg.br, diego.silva@gmail.com

[3] Center of Informatics, Federal University of Pernambuco
Av. Jornalista Anibal Fernandes, s/n - Cidade Universitária,
CEP 50.740-560, Recife, PE, Brazil
rmas@cin.ufpe.br

[4] LIAAD, Faculdade de Economia do Porto,
Rua Dr. Roberto Frias, s/n. 4200-464 Porto, Portugal
jfgoncal@fep.up.pt

[5] Internet and Network Systems Research, AT&T Labs Research
180 Park Avenue, Room C241, Florham Park, NJ 07932 USA
mgcr@research.att.com

[6] Dept. of Mathematics and Applications "R. Caccioppoli",
University of Napoli FEDERICO II
Compl. MSA, Via Cintia - 80126, Napoli, Italy
paola.festa@unina.it

Abstract. This paper presents a new approach to solve the NP-complete *minimum branch vertices problem* (MBV) introduced by Gargano et. al[1]. In spite of being a recently proposed problem in the network optimization literature, there are some heuristics to solve it [3]. The main contribution of this paper consists in a new heuristic based on the iterative refinement approach proposed by Deo and Kumar [2]. The experimental results suggest that this approach is capable of finding solutions that are better than the best known in the literature. In this work, for instance, the proposed heuristic found better solutions for 78% of the instances tested. The heuristic looks very promising for the solution of problems related with constrained spanning trees.

Keywords: Constrained spanning trees, Branch vertices, Iterative refinement.

1 Introduction

Given a undirected unweighted graph $G = (V, E)$ the *minimum branch vertices problem* (MBV) consists in finding the spanning tree of G which has the minimum

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 421–433, 2011.
© Springer-Verlag Berlin Heidelberg 2011

number of *branch* vertices [1]. A vertex v of G is said to be a *branch* vertex if its degree δ is greater than 2, i.e., $\delta(v) > 2$.

This problem has been recently proposed in the optimization literature. The main contributions were made by Cerulli et al. [3], who developed a mixed integer linear formulation which is able to find the optimal solution. However, for a reasonable computational running time, the model can only solve small instances. For large instances the authors proposed 3 heuristic methods capable of finding suboptimal solutions for the MBV: *Edge Weighting Strategy* (EWS), *Node Coloring Heuristic* (NCH), and a *combined strategy* (CS) between EWS and NCH. Details about these methods as well as their pseudo-codes can be found in [3].

The paper is organized as follows. In Section 2, we describe the iterative refinement algorithms introduced by Deo and Kumar [2]. In Section 3, we describe our iterative refinement algorithm for minimum branch vertices problem. Computational results are described in Section 4, and concluding remarks are made in Section 5.

2 Iterative Refinement and Constrained Spanning Trees

Among the approaches used in the literature to solve NP-complete constrained spanning tree problems there are the iterative refinement algorithms (IR) [2]. Consider the problem of constrained spanning tree defined by a weighted graph G and two constraints, $\mathcal{C}_1$ and $\mathcal{C}_2$, where $\mathcal{C}_1$ consists typically in the minimization of the sum of the weights in the spanning tree. The algorithm IR starts from a spanning tree partially constrained (which satisfies only $\mathcal{C}_1$) and moves at each iteration in the direction of a fully constrained tree (which satisfies $\mathcal{C}_2$), but sacrificing the optimality in relation to $\mathcal{C}_1$.

The general idea of the method is shown in the algorithm 1, extracted from [2]. First, a spanning tree T which satisfies only constraint $\mathcal{C}_1$ is constructed. Next, the edges which do not satisfy constraint $\mathcal{C}_2$ in T are identified and their weights in G are modified, originating G'. This is done in such a way that the new spanning tree constructed from G' violates less the constraint $\mathcal{C}_2$, in a step called *blacklisting*, whose aim is to discourage certain edges from reappearing in the next spanning trees. Usually the trick used in the *blacklisting* consists in increasing the weight of the edge associated with a violation of $\mathcal{C}_2$ in T. After each *blacklisting* step, a new spanning tree is constructed which satisfies only $\mathcal{C}_1$. This steps are repeated until a tree that satisfies $\mathcal{C}_2$ is found. Note that, the final spanning tree will satisfy $\mathcal{C}_2$, but will be sub-optimal in relation to $\mathcal{C}_1$.

The iterative refinement method is simple and easy to apply. The core of method consists in the design of a penalty function, or *blacklisting*, specific for the problem being studied. To be effective many important decisions must be taken regarding the number of edges to be penalized, what edges to penalize, and the value of the penalty for each edge to be penalized.

In their paper, Deo and Kumar [2] applied the IR method for the *Degree Constrained Minimum Spanning Tree* problem. The implemented algorithm alternates

the computation of the Minimum Spanning Tree (MST) with the increase of the weights on the edges whose degree exceeds a predetermined limit d.

Algorithm 1. `Iterative-Refinement-Algorithm(`G`, `$\mathcal{C}_1$`, `$\mathcal{C}_2$`)`

1. In graph G find a spanning tree that satisfies $\mathcal{C}_1$
2. **while** spanning tree violates $\mathcal{C}_2$ **do**
3. Using $\mathcal{C}_2$ alter weight of edges in G to obtain G' with new weights
4. In graph G' find a spanning tree that satisfies $\mathcal{C}_1$
5. Set $G \leftarrow G'$
6. **end while**

In the *blacklisting*, the edges are penalized by a quantity proportional to:

(i) the number $f[e]$ of degree-violating vertices where the edge e is incident;
(ii) a constant k defined by the user;
(iii) the weight $w[e]$ and the range of weights in current spanning tree, given by $w_{min} \leq w[e] \leq w_{max}$.

All the edges e incident to a degree violating vertex, except for the edge with smallest weight amongst them, are penalized as follows:

$$w'[e] = w[e] + kf[e]\left(\frac{w[e] - w_{min}}{w_{max} - w_{min}}\right)w_{max}. \tag{1}$$

In another paper, Boldon et. al [4] applied the dual-simplex approach to the *Degree Constrained Minimum Spanning Tree Problem* involving iterations in two stages until the convergence criteria are reached. The first stage consists in computing a MST using Prim's algorithm, which in the first iterations will violate the degree constraints of several vertices. The second stage consists in adjusting the weights of the violating edges using a *blacklisting* function which will increase the weight of an edge e as follows:

$$w'[e] = w[e] + fault \times w_{max} \times \left(\frac{w[e] - w_{min}}{w_{max} - w_{min}}\right). \tag{2}$$

In this function, $fault$ is a variable which takes the values 0, 1 or 2 depending on the number of vertices incident to the edge which is currently violating the degree constraint. Note that, this approach is very similar to the one used by Deo and Kumar [2]. The refinement idea is also referred in [5].

Other authors have applied the iterative refinement approach for other tree problems, such as *Diameter-Constrained Minimum Spanning Tree* [6]. In that paper, the authors presented two algorithms using the iterative refinement, IR1 and IR2. IR1 consists in iteratively computing a MST as solution to the tree diameter problem, and applying penalties to a subset of edges of the graph, such that they will be discouraged from appearing in the next iteration. The selection of the edges to modify is associated with presence of these edges or not in long paths of the tree, since its elimination aims at reducing the diameter of the tree.

Let L be a set of edges to be penalized; $w(l)$ the current weight of edge l; w_{max} and w_{min} the smallest and largest weight in the current spanning tree, respectively; $dist_c(l)$ the distance of the edge l to the central vertex of the path, increased by 1 unit. When the center is the edge l_c, we have $dist_c(l_c) = 1$, and as well there the only edge l incident to one of the extremes of the central edge l_c will have $dist_c(l) = 2$. The penalty imposed to each edge l in the current spanning tree will be:

$$\max\left\{ \left(\frac{w(l) - w_{min}}{dist_c(l)(w_{max} - w_{min})} \right) w_{max}, \epsilon \right\}, \quad (3)$$

where $\epsilon > 0$ is the minimum penalty which guarantees that the iterative refinement will not stay in the same spanning tree when the sum of the edges has penalty zero. The penalty decreases as the edges penalized are more distant from the center of current spanning tree, in such way that a path is broken in two sub-paths significantly shorter instead of a short sub-path and a long sub-path. The algorithm IR2 works almost same way as IR1, except for the fact that it does not recompute a new spanning tree at each iteration. A new spanning tree is created by modifying the current spanning tree, by removing one edge at a time.

3 An Iterative Refinement Algorithm for the MBV Problem

Section 2 presented several cases where the iterative refinement approach has been used to solve constrained spanning tree problems. Although these cases deal with weighted graphs, we propose an adaptation of the IR approach to solve the *Minimum Branch Vertices Problem*.

Let $G = (V, E)$ be a unweighted undirected graph representing a network in which we would like to find a spanning with the minimum number of vertices *branch*. By assigning random weights in the interval $[0, \ldots, 1]$ to each edge $e \in E$, the graph G becomes a new weighted graph $G' = (V, E)$. A minimum spanning tree T constructed on G' using the Kruskal's algorithm would be the starting solution for the iterative refinement algorithm. However, this initial solution may not satisfy the constraint $\delta(v) \leq 2, \forall\, v \in V$. Therefore, the topology of the initial solution will depend only on the weight that were assigned to the edges of G'.

Usually, the method starts with an initial solution with many vertices *branch*. The spanning tree T will then be modified iteratively, being recreated by changes to the topology of the previous spanning tree, and moves towards better solutions, i.e., with fewer vertices *branch*.

The difference between the iterative process proposed in this paper for the *minimum branch vertices* problem and the other iterative processes published by [2], [5], and [6] is the way the penalties are applied to the violating incident edges. In previous works the idea has been to penalize edges by increasing their weights to discourage them from reappearing in the trees in the next iterations. In this paper, we choose to penalize each violating edge by removing it explicitly

from the tree and replacing it with an edge with less violations. A violating edge of T (denoted by 'cutting edge') is selected for removal and is replaced by another edge of G' which is not yet in T (denoted by 'replacement edge'). Such replacement is defined by the exchange of the weights of the cutting edge and the replacement edge in G'. This replacement of edges continues until there are no replacement can reduce the number of vertices *branch* in the current tree T. The algorithm 2 describes the steps of the algorithm, and will be detailed next.

The strategy used to replace the edges is based on two measures of the violation of the edges, expresses as 1) the number of actual end vertices violating the edge (α); and 2) the sum of the degree of the extreme edges of the edge minus 2 (σ), which tells the sum of the degree of extremes of edge if we removed it from the tree. Good cutting edges are those edges that have many violating end vertices (i. e., with high values of α, followed by high values of σ). In a similar way, a good replacement edge is an edge that contributes the most to the reduction of the number of vertices *branch* in T (i.e., with low values of α, followed by low values of σ).

At each iteration of the refinement, one identifies a cutting edge to be removed from the tree T. Any edge incident to a vertex *branch* can be chosen as a cutting edge; edges of this type are used to construct a list of candidates L_{cut}. Next, we select one of the edges in the candidate list and remove it from L_{cut} and from T. The choice of the cutting edges at each iteration takes into account the degree of violation of the edge in T, quantified by the values α and σ. The edge selected will be the edge that has the largest α followed by the largest σ.

The removal of the cutting edge will divide the tree T into two connected components, and the set V of vertices of T into two sub-sets, S and S'. To avoid cycles, each edge removed from T is inserted into a special set, denoted by B_{list}, which indicates that the edges is tagged and cannot be reinserted in T.

To reconnect the two connected components we need to find an advantageous replacement edge capable of connecting both components without creating cycles. The candidate list L_{rep} includes all the edges in G which are not in T and that are capable of connecting both components and are not in B_{list}. A good replacement edge is an edge that does create violations when it is inserted in T, i.e., an edge that has lower values of α, followed by lower values of σ. Once the replacement edge selected is inserted in T, the current iteration of the algorithm ends.

If there is no advantageous replacement edges, then the replacement does not occur. The cutting edge returns to T. A new cutting edge is selected from L_{cut} and a new list L_{rep} is created to select the replacement edge. The algorithm continues until no more cutting edges can be replaced in T, i.e., $L_{cut} = \emptyset$.

We will illustrate the method using a simple instance of the problem MBV. The steps are shown in Figure 1. In (a) we have an initial spanning tree T, created by applying Kruskal's algorithm on graph G'. The tree shown in (a) is the result of assigning weights to the edges of G', which initial topology indicates the occurrence of two vertices *branch*: vertices 5 and 8.

Algorithm 2. `Mbv-Iterative-Refinement-Algorithm`$(G = (V, E))$

1. $G' \leftarrow$ `AssignRandomWeights`(G)
2. $T \leftarrow$ `CalculateMinimumSpanningTree`(G')
3. $B_{list} \leftarrow \emptyset$
4. **repeat**
5. `ThereWasExchange` $\leftarrow$ `false`
6. $L_{cut} \leftarrow$ `CreateCutList`(T, B_{list})
7. **while** ((`ThereWasExchange` $\neq$ `true`) $\wedge$ ($|L_{cut}| \neq 0$)) **do**
8. $(u^*, v^*) \leftarrow$ `SelectArcFromCutList`(L_{cut})
9. $L_{cut} \leftarrow L_{cut} \setminus \{ (u^*, v^*) \}$
10. $L_{rep} \leftarrow$ `CreateReplacementListToCutArc`$(T, G', (u^*, v^*))$
11. $(u, v) \leftarrow$ `SelectArcFromReplacementList`(L_{rep})
12. **if** $(\exists\ (u, v))$ **then**
13. $B_{list} \leftarrow B_{list} \bigcup \{(u, v)\}$
14. `SwapWeightsIntoGraph`$((u^*, v^*), (u, v))$
15. $T \leftarrow T \setminus \{(u^*, v^*)\}$
16. $T \leftarrow T \bigcup \{(u, v)\}$
17. `ThereWasExchange` $\leftarrow$ `true`
18. **end if**
19. **end while**
20. **until** (`ThereWasExchange` $\neq$ `false`)
21. **return** T

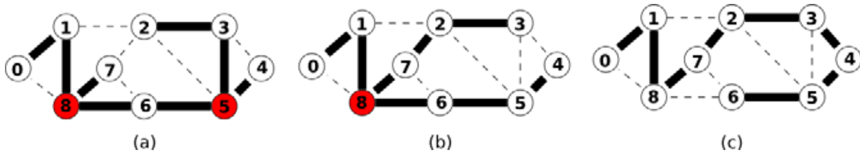


Fig. 1. An example of the iterative refinement algorithm solving an instance of MBV

In the sequence, the algorithm determines a cutting edge and a replacement edge and tries to transform T into a tree with less violations. L_{cut} is constructed with the incident edges to vertices *branch*, i.e.: $L_{cut} = \{(5,3), (5,4), (5,6), (8,1), (8,6), (8,7)\}$. The edges $(5,3)$, $(5,6)$, $(8,1)$ and $(8,6)$ have the highest values of α, followed by σ. We select edge $(5,3)$, which is removed from T and from L_{cut} and creates a cut in T which separates V in $S = \{2,3\}$ e $S' = \{0,1,4,5,6,7,8\}$.

The edges capable of connecting S and S' are $L_{rep} = \{(1,2), (2,5), (2,7), (3,4)\}$. Amongst these, the edges $(2,7)$ and $(3,4)$ are the only ones that can replace $(5,3)$ without causing any violations in the tree. We choose to replace $(5,3)$ by $(2,7)$, inserting $(2,7)$ in T to obtain (b). This concludes the first iteration of the algorithm.

The next iteration continues from the tree T showed in (b). $L_{cut} = \{(8,1), (8,6), (8,7)\}$, where $(8,1)$ is the most interesting edge to cut since it has the highest values of α and σ. We remove $(8,1)$ from T and L_{cut}, creating the cut $S = \{0,1\}$ and $S' = \{2,3,4,5,6,7,8\}$. The edges capable of connecting S and S' are $L_{rep} = \{(0,8), (1,2)\}$, which have the same value of α and σ. Edge $(0,8)$ is selected to replace $(8,1)$ in T.

Replacing edge $(8, 1)$ by $(0, 8)$ does not bring any advantages in T since it does not reduce the number of vertices *branch* in the tree, which will continue having 1 vertex *branch*. The replacement is canceled, edge $(8, 1)$ returns to the tree T, and we select a new cutting edge from $L_{cut} = \{(8, 6), (8, 7)\}$. The edge selected is $(8, 6)$, given the value of α and σ. Removing $(8, 6)$ from the tree will divide V into $S = \{0, 1, 8, 7, 2, 3\}$ and $S' = \{6, 5, 4\}$, with $L_{rep} = \{(2, 5), (3, 4), (6, 7)\}$. The edge $(3, 4)$ is the best replacement for edge $(8, 6)$, and is inserted into T. The replacement ends the second iteration of the algorithm, resulting in tree (c).

The third iteration begins with $L_{cut} = \emptyset$. There are no more vertices *branch* in the tree and therefore there is no cutting edge available. The tree T presented in (c) is then a solution to be returned by the algorithm.

4 Experimental Results

In this section, we present results on computational experiments with the iterative refinement method applied to a set of instances with the purpose of comparing the quality of the results obtained by our IR algorithm with the results obtained by the heuristics EWS and NCH proposed by [3]. All the algorithms cited were implemented in `ANSI C++`, compiled with `gcc version 4.3.2`, using the libraries `STL` and run in the operating system `Ubuntu 4.3.2-1`. The algorithm used to find an initial solution was the Kruskal's algorithm, using efficient data structures to represent the disjoint sets (*union-find structures*). This data structures where used by all methods to determine if two vertices were in different connected components of a graph, as well as to determine the replacement edges candidates capable of connecting S e S'. Details about the efficient implementation of the data structures *union-find* can be found in [7] and [8].

The instances were created by the network flow problem generator NetGen [9], available in `public ftp` from DIMACS [1]. The instances were divided into different classes, each containing different number of vertices and edges. NetGen constructs network flow problems using as input a file that specifies the input parameters. Since the *minimum branch vertices* problem consists of an unweighted and uncapacitated graph, we used only the topology of the graphs input. Repeated edges were ignored.

The input files used by NetGen to generate the instances follow the format given in Table 1. According to the table, the only parameters that can vary are the seed for the random number generator and the number of vertices and edges of the output graph. In tables 2 and 3 the values presented for each instance in columns d, n and s, correspond to the number of edges, number of vertices, and seed for each instance, respectively. The column m represents the 'real' number of edges of the graph, removing the repeated edges.

We have generated instances with 30, 50, 100, 150, 300, and 500 vertices, with edges densities of 15% and 30% in 5 graphs capable of representing each of these classes.

[1] `ftp://dimacs.rutgers.edu/pub/netflow/`

Table 1. NetGen parameters for input files

#	*Parameters*	*Input*	*Parameter Description*
1	SEED	*Variable*	Random numbers seed
2	PROBLEM	1	Problem number
3	NODES	*Variable*	Number of nodes
4	SOURCES	1	Number of sources (including transshipment)
5	SINKS	1	Number of sinks (including transshipment)
6	DENSITY	*Variable*	Number of (requested) edges
7	MINCOST	0	Minimum cost of edges
8	MAXCOST	1000	Maximum cost of edges
9	SUPPLY	1	Total supply
10	TSOURCES	0	Transshipment sources
11	TSINKS	0	Transshipment sinks
12	HICOST	1	Percent of skeleton edges given maximum cost
13	CAPACITED	1	Percent of edges to be capacitated
14	MINCAP	0	Minimum capacity for capacitated edges
15	MAXCAP	3	Maximum capacity for capacitated edges

The methods EWS and NCH were run only once, since the runs result in the same deterministic values. The iterative refinement method has been statistically evaluated, since it depends on the weights of the graph G' assigned randomly at the beginning of the algorithm. The methodology used consisted in 100 runs for each instance, each one with a different seed. Tables 2 and 3 present the minimum, maximum, average, median, standard deviation, and variance found for the execution time, and the solution value of each instance, respectively in columns 'Min', 'Max', 'Mean', 'Med', 'Dev' and 'Var' of the column 'Value' corresponding to the results of algorithm IR.

The rows of column 'C' are tagged with the character 'y' when the IR methods found solutions with an average number of vertices *branch* (column 'Mean') lower than the values found by the algorithms EWS and NCH. This condition occurred happens in 43 out 55 instances (78% of the instances).

The median values suggests that the IR method performed very well for these instances. For the 55 instances tested, the IR method obtained median values better than the ones obtained by EWS and NCH in 37 instances, and equal values in 15 instances.

Even when the IR method did not obtain the best values, we can see that it obtains value very close to the ones obtained by EWS and NCH.

The histograms in Figures 2 – 7 report frequencies computed for the 100 runs of some of the instances in which IR did not obtain better values than EWS and NCH. Note that, the most frequent strip corresponds to values close or equal to the EWS and NCH heuristics.

The reported running times are in seconds. The running times of the EWS and NCH heuristics correspond to only one run. The running time for the IR algorithm varies since the computational effort to find a solution depends on the

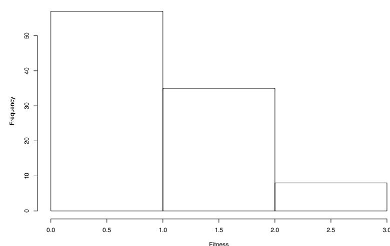


Fig. 2. Histogram for the instance $n = 30$, $m = 68$, $s = 7236$: $NCH_{val} = 1$; $IR_{mean} = 1,37$; $freq[0\dots1] \sim 55$

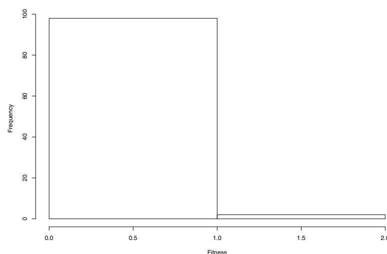


Fig. 3. Histogram for the instance $n = 30$, $m = 135$, $s = 5081$: $NCH_{val} = 0$; $IR_{mean} = 0,24$; $freq[0\dots1] \sim 98$

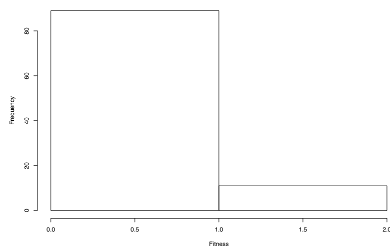


Fig. 4. Histogram for the instance $n = 50$, $m = 375$, $s = 1720$: $NCH_{val} = 0$; $IR_{mean} = 0,56$; $freq[0\dots1] \sim 90$

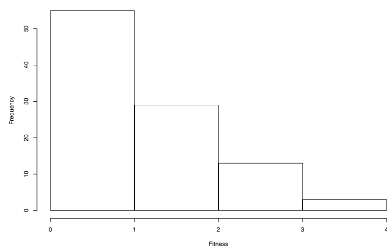


Fig. 5. Histogram for the instance $n = 100$, $m = 750$, $s = 5885$: $NCH_{val} = 1$; $IR_{mean} = 1,5$; $freq[0\dots1] \sim 55$

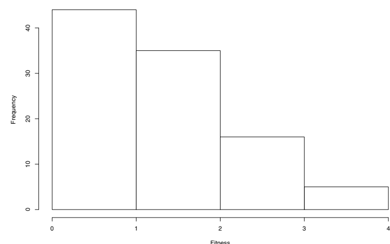


Fig. 6. Histogram for the instance $n = 150$, $m = 1688$, $s = 3738$: $NCH_{val} = 1$; $IR_{mean} = 1,69$; $freq[0\dots1] \sim 45$

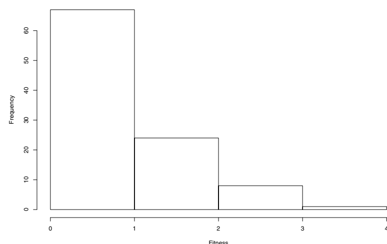


Fig. 7. Histogram for the instance $n = 300$, $m = 6750$, $s = 4889$: $NCH_{val} = 1$; $IR_{mean} = 1,27$; $freq[0\dots1] \sim 65$

number of *branch* vertices existing in the initial solution, and therefore should be analyzed statistically by the measures 'Min', 'Max', 'Mean', 'Dev' and 'Var' of the block 'Time'. These measure correspond to the minimum, the maximum, the average, the median, the standard deviation, and variance of the 100 runs of the IR algorithm, respectively.

Table 2. Comparison results of 'value' and 'running time' for EWS, NCH, and IR

Instances				**Cerulli et al**				**Iterative Refinement Approach**											
				EWS		**NCH**		**Value**							**Time**				
d	*n*	*m*	*s*	*Val*	*Time*	*Val*	*Time*	*Min*	*Mean*	*Med*	*Max*	*Dev*	*Var*	*C.*	*Min*	*Mean*	*Max*	*Dev*	*Var*
68	**30**	**67**	**1596**	**2**	**0,004**	**2**	**0,004**	**0**	**0,85**	**1**	**3**	**0,7**	**0,49**	**y**	**0,000**	**0,002**	**0,012**	**0,003**	**0,000**
68	**30**	**67**	**2429**	**2**	**0,008**	**2**	**0,008**	**0**	**0,68**	**1**	**3**	**0,71**	**0,5**	**y**	**0,000**	**0,002**	**0,008**	**0,002**	**0,000**
68	**30**	**66**	**7081**	**2**	**0,008**	**2**	**0,004**	**0**	**1,11**	**1**	**3**	**0,9**	**0,81**	**y**	**0,000**	**0,003**	**0,008**	**0,003**	**0,000**
68	30	66	7236	1	0,012	1	0,008	0	1,37	1	3	0,82	0,68	**n**	0,000	0,003	0,008	0,003	0,000
68	30	66	7880	1	0,012	1	0,012	0	1,37	1	3	0,77	0,6	**n**	0,000	0,002	0,008	0,002	0,000
135	**30**	**124**	**1172**	**1**	**0,020**	**1**	**0,012**	**0**	**0,84**	**1**	**2**	**0,65**	**0,42**	**y**	**0,000**	**0,004**	**0,016**	**0,003**	**0,000**
135	30	122	2488	0	0,028	0	0,004	0	0,4	0	2	0,55	0,3	**n**	0,000	0,004	0,012	0,003	0,000
135	**30**	**122**	**4970**	**1**	**0,016**	**1**	**0,020**	**0**	**0,45**	**0**	**2**	**0,54**	**0,29**	**y**	**0,000**	**0,004**	**0,012**	**0,003**	**0,000**
135	30	128	5081	0	0,036	0	0,008	0	0,24	0	2	0,47	0,22	**n**	0,000	0,003	0,012	0,003	0,000
135	**30**	**125**	**8788**	**1**	**0,032**	**1**	**0,016**	**0**	**0,28**	**0**	**1**	**0,45**	**0,2**	**y**	**0,000**	**0,004**	**0,016**	**0,003**	**0,000**
188	**50**	**182**	**1054**	**2**	**0,064**	**2**	**0,028**	**0**	**1,55**	**1**	**5**	**1,08**	**1,16**	**y**	**0,000**	**0,008**	**0,020**	**0,004**	**0,000**
188	**50**	**179**	**3335**	**2**	**0,056**	**2**	**0,028**	**0**	**1,16**	**1**	4	**0,73**	**0,54**	**y**	**0,000**	**0,009**	**0,024**	**0,004**	**0,000**
188	**50**	**180**	**4663**	**2**	**0,052**	**3**	**0,024**	**0**	**1,12**	**1**	4	**0,79**	**0,63**	**y**	**0,000**	**0,008**	**0,024**	**0,005**	**0,000**
188	**50**	**182**	**4985**	**2**	**0,060**	**2**	**0,024**	**0**	**1,5**	**1**	4	**0,92**	**0,84**	**y**	**0,000**	**0,008**	**0,020**	**0,004**	**0,000**
188	**50**	**186**	**7085**	**4**	**0,056**	**4**	**0,028**	**0**	**1,39**	**1**	**3**	**0,84**	**0,7**	**y**	**0,000**	**0,008**	**0,016**	**0,004**	**0,000**
375	50	341	1720	0	0,132	0	0,048	0	0,56	0	2	0,69	0,47	**n**	0,004	0,012	0,024	0,005	0,000
375	**50**	**345**	**6752**	**2**	**0,164**	**2**	**0,048**	**0**	**0,36**	**0**	**3**	**0,58**	**0,33**	**y**	**0,004**	**0,013**	**0,024**	**0,005**	**0,000**
375	**50**	**349**	**7009**	**2**	**0,148**	**2**	**0,040**	**0**	**0,42**	**0**	**2**	**0,59**	**0,35**	**y**	**0,004**	**0,012**	**0,020**	**0,004**	**0,000**
375	**50**	**343**	**7030**	**1**	**0,160**	**1**	**0,040**	**0**	**0,32**	**0**	**2**	**0,51**	**0,26**	**y**	**0,000**	**0,012**	**0,020**	**0,004**	**0,000**
375	50	344	9979	0	0,144	0	0,056	0	0,4	0	2	0,62	0,38	**n**	0,004	0,012	0,020	0,004	0,000
750	**100**	**723**	**2312**	**3**	**0,588**	**3**	**0,292**	**0**	**1,28**	**1**	**3**	**0,94**	**0,89**	**y**	**0,024**	**0,046**	**0,080**	**0,011**	**0,000**
750	**100**	**730**	**299**	**3**	**0,708**	**3**	**0,248**	**0**	**1,09**	**1**	4	**0,95**	**0,91**	**y**	**0,028**	**0,046**	**0,072**	**0,010**	**0,000**
750	**100**	**722**	**4414**	**2**	**0,544**	**2**	**0,244**	**0**	**1,41**	**1**	4	**0,98**	**0,95**	**y**	**0,024**	**0,044**	**0,068**	**0,010**	**0,000**
750	100	724	5885	1	0,640	1	0,184	0	1,5	1	4	0,99	0,98	**n**	0,024	0,046	0,084	0,010	0,000
750	**100**	**719**	**6570**	**3**	**0,644**	**3**	**0,244**	**0**	**1,69**	**2**	**5**	**1,12**	**1,25**	**y**	**0,028**	**0,046**	**0,084**	**0,011**	**0,000**

Table 3. Comparison results of 'value' and 'running time' for EWS, NCH. and IR.

Instances				Cerulli et al				Iterative Refinement Approach												
				EWS		NCH		Value							Time					
d	*n*	*m*	*s*	*Val*	*Time*	*Val*	*Time*	*Min*	*Mean*	*Med*	*Max*	*Dev*	*Var*	*C.*	*Min*	*Mean*	*Max*	*Dev*	*Var*
1500	100	1399	5309	1	2,276	1	0,500	0	0,55	0	2	0,66	0,43	y	0,040	0,082	0,128	0,017	0,000
1500	100	1383	6105	1	1,820	1	0,372	0	0,43	0	2	0,59	0,35	y	0,040	0,076	0,128	0,015	0,000
1500	100	1386	6259	1	2,112	1	0,316	0	0,4	0	2	0,57	0,32	y	0,040	0,077	0,112	0,015	0,000
1500	100	1389	7695	1	2,096	1	0,380	0	0,34	0	2	0,54	0,29	y	0,036	0,074	0,112	0,015	0,000
1500	100	1391	9414	0	2,348	0	0,500	0	0,66	1	3	0,71	0,51	n	0,056	0,083	0,132	0,017	0,000
1688	150	1624	199	3	2,684	2	0,620	0	2,06	2	6	1,25	1,55	n	0,092	0,146	0,208	0,024	0,001
1688	150	1619	3738	1	3,060	1	0,872	0	1,69	2	4	1,05	1,1	n	0,096	0,140	0,264	0,024	0,001
1688	150	1624	5011	4	3,696	3	0,932	0	1,52	1,5	4	1,03	1,06	y	0,072	0,135	0,200	0,024	0,001
1688	150	1627	7390	2	2,804	2	0,700	0	1,62	1,5	5	1,1	1,21	y	0,068	0,146	0,244	0,035	0,001
1688	150	1624	878	3	3,288	2	0,884	0	1,82	2	5	1,11	1,24	y	0,076	0,147	0,272	0,040	0,002
3375	150	3120	2051	1	9,405	1	1,572	0	0,46	0	2	0,58	0,33	y	0,200	0,300	0,424	0,062	0,004
3375	150	3120	2833	1	9,189	1	1,288	0	0,5	0	2	0,58	0,33	y	0,204	0,303	0,432	0,062	0,004
3375	150	3141	3064	1	11,089	1	1,620	0	0,58	0	3	0,68	0,47	y	0,208	0,330	0,436	0,062	0,004
3375	150	3116	5357	1	10,025	1	1,688	0	0,29	0	2	0,48	0,23	y	0,192	0,292	0,416	0,054	0,003
3375	150	3117	5687	2	10,933	2	1,588	0	0,34	0	2	0,54	0,29	y	0,164	0,292	0,428	0,058	0,003
6750	300	6502	1545	1	45,391	1	5,720	0	1,62	2	4	1,07	1,15	n	0,724	1,042	1,332	0,116	0,013
6750	300	6471	365	3	43,331	3	6,144	0	1,81	2	5	1,01	1,02	y	0,884	1,054	1,444	0,108	0,012
6750	300	6481	4071	5	45,731	3	5,888	0	1,61	1,5	5	1,13	1,27	y	0,852	1,071	1,432	0,116	0,013
6750	300	6513	4889	1	45,303	1	5,832	0	1,27	1	4	0,86	0,74	n	0,852	1,034	1,324	0,114	0,013
6750	300	6505	681	4	45,239	4	5,696	0	1,88	2	5	0,99	0,98	y	0,868	1,056	1,444	0,116	0,013
13500	300	12539	1358	2	151,349	2	11,273	0	0,54	0	3	0,64	0,41	y	2,232	3,004	3,668	0,349	0,122
13500	300	12508	2067	3	160,454	2	13,337	0	0,34	0	3	0,55	0,31	y	2,460	3,074	3,748	0,263	0,069
13500	300	12447	4372	1	178,551	1	13,541	0	0,4	0	2	0,6	0,36	y	2,464	3,250	3,808	0,304	0,092
13500	300	12480	960	1	179,371	1	13,513	0	0,65	1	3	0,67	0,45	y	2,372	2,996	3,716	0,333	0,111
13500	300	12474	9886	1	181,843	1	13,369	0	0,49	0	3	0,69	0,47	y	1,740	2,939	3,772	0,453	0,206
18750	500	18034	1456	2	290,798	2	24,350	0	1,85	2	4	1,12	1,26	y	4,924	5,665	8,073	0,716	0,513
18750	500	18055	1653	3	279,593	3	23,586	0	1,4	1	4	1,03	1,07	y	4,860	6,188	8,129	0,853	0,727
18750	500	18009	4444	2	299,959	2	29,026	0	1,74	2	5	1,05	1,1	y	4,832	6,678	8,161	0,924	0,853
18750	500	18048	6849	2	313,808	2	29,798	0	1,81	2	5	1,06	1,13	y	4,912	6,833	8,181	0,913	0,833
18750	500	18037	8824	4	320,636	3	29,142	0	1,59	2	4	0,99	0,97	y	4,776	6,596	7,945	0,893	0,797

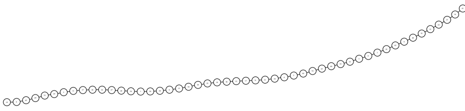


Fig. 8. IR solution for the inst. $n = 50$, $m = 186$, $s = 7085$, $branch = 0$

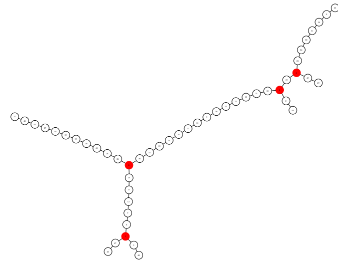


Fig. 9. NCH solution for the inst. $n = 50$, $m = 186$, $s = 7085$, $branch = 4$

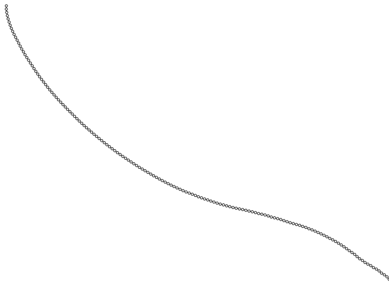


Fig. 10. IR solution for the inst. $n = 150$, $m = 1688$, $s = 5011$, $branch = 0$

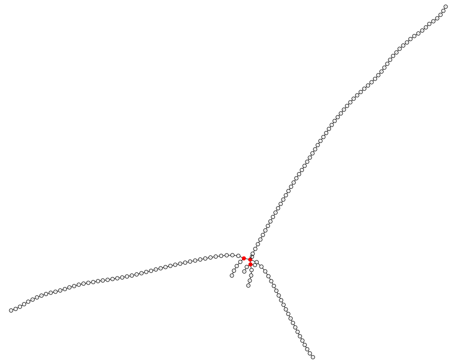


Fig. 11. NCH solution for the inst. $n = 150$, $m = 1688$, $s = 5011$, $branch = 3$

It is worth mentioning that for each instance from *benchmark* evaluated, there was at least one run of the 100 runs were the IR method obtained a solution with zero vertices *branch*. Figures 8, 9, 10, and 11 depict the difference in topology of some of the best solutions found by IR and the one found by NCH. The vertices highlighted correspond to vertices *branch*, i.e., $\delta(v) > 2$.

5 Concluding Remarks

According to the results presented in Section 4, for the *benchmark* used in the paper, the iterative method presented has better performance than the methods proposed by [3]: edge weighting and node coloring strategies. In 78% of the instances the IR algorithm obtained average results better than the ones found by the methods EWS and NCH. The small standard deviation as well as the better median values in 37 of the 55 instances classes further support quality of the IR algorithm compared to EWS and NCH.

The experimental results show that the iterative refined method is an effective approach to solve the MBV directly or as a sub-problem of large problems. Since the test *benchmark* is made of artificial instances generated by NetGen, further research should be conducted using real instances. Comparisons with exact methods such as the algorithms proposed in [3] should also be carried in future experiments.

Acknowledgments

Ricardo M.A Silva was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq), the Fundation for Support of Research of the State of Minas Gerais, Brazil (FAPEMIG), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brazil (CAPES), and Fundação de Apoio ao Desenvolvimento da UFPE, Brazil (FADE). José F. Gonçalves was partially supported by Fundação para a Ciência e Tecnologia (FCT) project PTDC/GES/72244/2006. Diego M. Silva was partially supported by CAPES-MINTER Program between the Federal Universities of Minas Gerais and Lavras, Brazil.

References

1. Gargano, L., Hell, P., Stacho, L., Vaccaro, U.: Spanning trees with bounded number of branch vertices. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 355–365. Springer, Heidelberg (2002)
2. Deo, N., Kumar, N.: Computation of Constrained Spanning Trees: A Unified Approach. In: Network Optimization. Lecture Notes in Economics and Mathematical Systems, vol. 450, pp. 194–220. Springer, New York (1997)
3. Cerulli, R., Gentili, M., Iossa, A.: Bounded-Degree Spanning Tree Problems: Models and New Algorithms. Comput. Optim. Appl. 42, 353–370 (2009)
4. Boldon, B., Deo, N., Kumar, N.: Minimum-Weight Degree-Constrained Spanning Tree Problem: Heuristics and Implementation on an SIMD Parallel Machine. Technical Report CS-TR-95-02, Department of Computer Science, University of Central Florida, Orlando, FL (1995)
5. Mao, L.J., Deo, N., Lang, S.D.: A Comparison of Two Parallel Approximate Algorithms for the Degree-Constrained Minimum Spanning Tree Problem. Congressus Numerantium 123, 15–32 (1997)
6. Abdalla, A., Deo, N., Gupta, P.: Random-Tree Diameter and the Diameter Constrained MST. Congressus Numerantium 144, 161–182 (2000)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn., pp. 498–509. MIT Press, Cambridge (2001)
8. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications, pp. 521–522. Prentice-Hall, Englewood Cliffs (1993)
9. Klingman, D., Napier, A., Stutz, J.: NETGEN – A program for generating large scale (un)capacitated assignment, transportation, and minimum cost flow network problems. Managent Science 20, 814–821 (1974)

Generating Time Dependencies in Road Networks

Sascha Meinert and Dorothea Wagner

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
{Sascha.Meinert,Dorothea.Wagner}@kit.edu

Abstract. In the last decade major progress has been made in accelerating shortest path queries in large-scale, time-dependent road networks. Most techniques are heuristically motivated and their performance is experimentally evaluated on real-world data. However, to our knowledge no free time-dependent dataset is available to researchers.

This is the first work proposing algorithmic approaches for generating time-dependent road networks that are built on top of static road networks in the scenario of systematic delays. Based on an analysis of a commercial, confidential time-dependent dataset we have access to, we develop algorithms that utilize either road categories or coordinates to enrich a given static road network with artificial time-dependent data. Thus, the static road-networks we operate on may originate from manifold sources like commercial, open source or artificial data. In our experimental study we assess the usefulness of our algorithms by comparing global as well as local statistical properties and the shortest-path structure of generated datasets and a commercially used time-dependent dataset. Until now, evaluations of time-dependent routing algorithms were based on artificial data created by ad-hoc random procedure. Our work enables researchers to conduct more reasonable validations of their algorithms than it was possible up to now.

1 Introduction

In recent years the focus of accelerating shortest path queries switched from static to time-dependent scenarios. One reason is that the time-dependent scenario is more realistic in the sense that the time needed to travel a certain distance depends on the traffic and thus changes over the day. The speedups for route calculations gained in this field are yet not as impressive as they are for static scenarios. Hence, many people are still attracted by this field of research to achieve similar results for time-dependent scenarios. To prove the applicability of new algorithms they are usually evaluated on large-scale, i.e., continental-size, time-dependent road networks. Ideally, this is done using real-world data, as most of the algorithms are custom tailored for road networks. Unfortunately, to our knowledge no freely available real-world dataset of time-dependent road networks exist. The reason may be that observing traffic and maintaining statistics is an expensive and tedious task [1]. Hence, companies do not share their valuable data. Thus, experimental evaluations are done using artificial datasets [2,3,4]. The test data used there is generated by an ad-hoc method for randomized delay assignment and it is not geared towards any properties of real-world data. We aim at closing this gap by

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 434–446, 2011.
© Springer-Verlag Berlin Heidelberg 2011

artificially generating time-dependency information by exploiting structural properties of an underlying static road network.

Related Work. In recent years, many publications on point-to-point route-planning techniques have appeared. Good overviews for the static [5] as well as the time-dependent scenario [6] exist. Time-dependent data is usually modeled by a mapping of functions to edges that determine for each time of the day how fast one can traverse a certain distance in relation to the default time when no delay occurs. These functions are called *profiles*. To our knowledge, the only known approach for artificially assigning time-dependent information on edges in a large scale road-network is to randomly assign delays on a specific level within the road network's inherent hierarchy [3]. The work does not provide a systematic analysis of the generated data. In a personal discussion with the authors it turned out that way too many profiles are generated, which have to be removed by an unreported process. The information on the shape of profiles is taken from the research field of traffic and transportation prognosis [7]. The simulations used there are not applicable to large-scale road networks of continental-size, e.g., Europe, for two reasons: First, no dataset exists that contains the travel behavior of all people living on a continent. Second, no algorithm is capable of simulating all individuals' behaviors within a road-network of that size [8].

Contribution. This is the first work providing algorithms which allow researchers to enrich static road-networks with time-dependent information and conduct more reasonable validations of their algorithms. Our scenarios are systematic time dependencies in road networks that occur on a daily basis. We explicitly do not cover dynamic time dependencies, i.e., delays occurring because of unexpected events like accidents.

The analysis of a commercial and confidential time-dependent small subset of the European road-network[1], namely Germany, gives insights on the properties of delays. Based upon these results we develop our algorithms. In order to generate meaningful time-dependent data, we either require the road network to have road-category information attached to edges, or to contain the coordinates of the nodes. Thus, our algorithms can be applied to graphs of manifold origin, e.g. commercial, open source or artificial. We use the additional data to compute structural information, in particular, areas of urban type. Based on this, we find edge candidates for assigning time-dependent information. In an extensive computational study, we show that the generated data has in global as well as local scope statistical properties similar to those of the real-world road network. Additionally, we show that the shortest-path structure exhibits similar characteristics on artificial and real-world data. Hence, a shortest-path technique that performs well on our generated datasets is likely to behave similarly on real-world data.

Outline. In Section 2 we establish the basis of our work by analyzing confidential commercial time-dependent road-network data[1]. We present statistical data and representative extracts of the time-dependent data and derive a systematic model of how delays occur. Based on our observations we develop our algorithms for generating time-dependency within static road networks in Section 3. In Section 4 we compare the results of our generators with real-world data, in order to demonstrate the usefulness of our algorithms.

[1] Provided by the company PTV (`http://www.ptv.de`.)

2 Analysis of Real-World Time-Dependent Data

The time-dependent road-network dataset of Germany[1] we use is confidential and part of commercial products. On the technical side, the dataset consists of two parts. The basis is a static network relying on data of the company NavTeq. The data is not fully free for scientific use but has been provided to participants of the 9th Dimacs Implementation Challenge [9]. We have access to a slightly updated version of the year 2006.

The second part consists of the time-dependency information. Delays are modeled by assigning profiles to edges. A widely applied solution is to use *piece-wise linear functions* (PWLF) to model the delay of specific intervals of the day. Intermediate points are interpolated linearly. The real-world dataset provides sets of PWLFs for different days of the week. A mapping assigns edges of the static dataset that are affected by a delay to the corresponding PWLF. Each day is split up into 96 time intervals of 15 minutes each. Accordingly, a PWLF consists of 96 support points. Each point represents the delay of traversing this edge as a factor in relation to the mean speed usually assigned. For the remaining part of this work we refer to these PWLFs as *profiles*.

The dataset contains *systematic* time-dependency profiles, i.e., it contains historical data of a larger, specific period of time that represents time-dependency information that is used commercially for daily routing. It explicitly does not cover dynamic time-dependency information, which may be caused by unpredictable events like accidents, holiday traffic or bad weather. The time-dependent data covers only a fraction of the static graph. Table 1 shows how many edges are affected by time-dependency. Additionally, the number of affected edges is broken down into their assigned road-categories. The first row of Table 1 shows that each of the daily time-dependent datasets consists of a few hundred profiles, each of which represents traveling times of hundreds of thousands of edges. Evidently, the data is already highly compressed. We push the compression idea to the limit to see whether a small set of representative profiles emerges.

Each profile is represented by a PWLF having 96 supporting points. We find similar profiles using the *k-means* algorithm of Lloyd [10]. To compare the profiles, we interpret them as points in $\mathbb{R}^{96}$. Since the outcome of k-means depends on the randomly chosen profiles in the initialization phase, the algorithm is restarted several times and the result with the lowest total distance is used. The results were stable over several executions of the k-means algorithm. Figure 1 shows all profiles of the days Tuesday to Thursday on the left side and the normalized ($[0.0, 1.0]$), k-means compressed profiles with $k = 4$ on the right; larger values of k gives similar functions differing only by an offset. Following the intuition, four types of profiles can be distinguished:

Table 1. Fraction of edges of the static graph, consisting of $\sim$11 million edges, affected by time-dependent profiles. The number of affected edges has been broken down into their road category.

	Mo	Tu-Th	Fr	Sa	So
#Profiles	406	436	420	255	153
% of tdEdges	4.60%	4.73%	4.30%	2.67%	1.74%
Expressway	14.05%	14.24%	12.90%	7.43%	10.12%
Non-urban	60.52%	60.44%	60.38%	66.75%	63.49%
Urban	25.43%	25.32%	25.72%	25.82%	26.39%

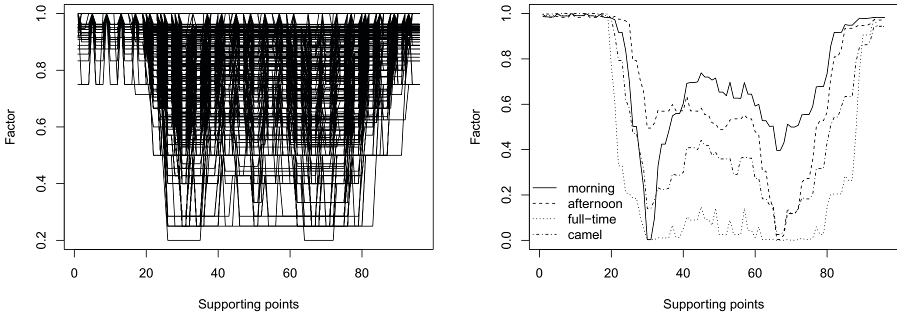


Fig. 1. On the left hand side all profiles for the days Tuesday to Thursday are shown. On the right hand side, the centroids of the first normalized ($[0.0, 1.0]$) and then compressed set of profiles for the days Tuesday to Thursday using the *k-means* algorithm with $k = 4$ is shown.

– **Full-time:** the travel speed is constantly reduced on a large part of the day (*FUL*).
– **Camel:** the travel speed is reduced in the morning and in the afternoon. In between the edge can be traversed faster (*CAM*).
– **Morning:** the travel speed is mainly reduced in the morning. Probably it is reduced in the afternoon too, but not as strong as in the morning (*MOR*).
– **Afternoon:** the travel speed is opposite to the case in the morning, thus, the main reduction is in the afternoon (*AFT*).

An explanation can be found in the structure of the underlying graph. Roads are modeled using undirected edges, where possible, to reduce space consumption, e.g., small roads that can be traversed in both directions. However, in the case of roads whose directional lanes are separated, directed edges are used. Thus, sections of roads that would have a camel profile assigned have a morning profile assigned for the one direction and the afternoon profile for the other direction.

A further, more detailed look at the road-network infrastructure reveals that urban regions mostly have full-time profiles assigned. Roads leading into urban regions have camel profiles assigned. In the case of separated lanes morning profiles are assigned to roads leading into urban regions whereas afternoon profiles are assigned to roads leaving urban regions.

So far, we made three observations that will help us in developing algorithms to create realistic time-dependent road networks: 1. Delays seem to originate from traffic between towns and their area of influence. 2. The profiles assigned to edges are different, based on their location within, leading into or leading out of town. 3. Profiles have a similar curve progression and are therefore well compressible. That way we can store a few representative profile types $P = \{FUL, CAM, MOR, AFT\}$ and use them for the assignment. To randomize the generated data and make it look more realistic, the profiles can be disturbed accordingly before they are assigned to edges. A fourth observation can be made that originates from the modeling process. Usually, roads have bends and crossings. Thus, they are often modeled having many edges and nodes. On the one hand it is important to know which edges are affected by time-dependency in general. On the other hand it is unlikely that profiles of adjacent nodes differ significantly. Thus, similar profiles should be assigned to paths of adjacent nodes.

3 Algorithms for Assigning Profiles

As already stated, real-world road networks are hard to obtain. Nevertheless, some sources for *static* scenarios are available, e.g., commercial data of PTV AG [9], artificially generated data [11] or open source data like the OpenStreetMap project[2]. Usually, these datasets not only contain nodes, edges and travel times of the underlying road network, but contain additional information. We exploit this information to locate urban regions within a given road network.

Real-world datasets often contain *road categories* that roughly correspond to common street categories, e.g., urban, express- or motorway, but are much more detailed. Each edge $e \in E$ has a single road category from the set of all road categories ζ assigned by a function $cat : E \rightarrow \zeta$. In some cases, the datasets may not contain road-category information in high detail, the data is flawed or is completely absent, e.g., when dealing with artificial road-networks. However, in many cases these datasets have in common that they rely on an embedding in the plane and thus contain *coordinates* for nodes. Each of the vertices $v \in V$ has coordinates assigned by a function $p : V \rightarrow \mathbb{R}^2$.

This additional information is used to compute structural information, which is then exploited to locate edges that are likely to be affected by time dependency. Note that we do not assign a profile to an edge immediately but rather attach profile types and their quantity to affected edges. This allows edges to be affected by different types of categories, e.g., if a road can be traversed in both directions it may get a morning as well as an evening profile type attached. Our algorithms work in five stages as follows.

Preprocessing. First, in the *preprocessing phase* structural data is computed to split the set of nodes V of the road network into two disjoint sets. Namely, *urban nodes* $U \subseteq V$ and *rural nodes* $R \subseteq V$. *Boundary nodes* are defined by the subset of urban nodes that are adjacent to at least one rural node, i.e., $B := \{u \in U \mid \exists (u,v) \in E, v \in R\}$. The following phases rely on this separation for profile type assignment. In some cases, the urban regions can be classified to form a set of disjoint towns.

Urban. Next, the urban phase starts. We assume that people who live within an urban region will travel only within an urban region. Thus, edges of paths that are used often will get a full-time profile type assigned.

Rural. In the *rural phase*, we determine the edges that are likely to have a morning or evening profile assigned. The basic idea is that commuters drive from rural into urban regions in the morning and back in the evening. This is not done arbitrarily. Each urban region has a surrounding region of influence denoted by *urban catchment*.

Filtering. In the case that too many edges are affected by profile types, these can be filtered to fit statistical properties, e.g., the profile distribution of the commercial data. We follow a simple rule: If less than a filter limit F profiles types are assigned to an edge, the edge is considered to be not affected by any delay.

Postprocessing. So far profile types $p \in P$ have been assigned to edges. In a more realistic scenario, these could now be created with random offsets or overlayed with each

[2] `http://www.openstreetmap.org/`

other. To preserve comparability we will omit this randomization part. Instead, each edge is examined and the profile type occurrences will determine the resulting profile type of the edge using the following mapping (! indicates no occurrance):
1. $(AFT \wedge !MOR) \rightarrow AFT$; 2. $(!AFT \wedge MOR) \rightarrow MOR$; 3. $(AFT \wedge MOR) \rightarrow CAM$; 4. $(FUL \wedge !AFT \wedge !MOR) \rightarrow FUL$. Note that other goals of this phase could be to make the transitions from exterior profiles to interior profiles smooth by interpolation, as soon as the boundary nodes are crossed. Furthermore, it may be interesting to overlay profiles on affected edges, e.g., to reduce harsh changes in the profiles. Problems arising when dealing with both of these problems are not covered here.

In the following we present three algorithmic approaches to find edges that are likely to be affected by time-dependent profiles types without having any traffic information available. The phases of each algorithm are described in detail. Note that all of the algorithms use the filtering and postprocessing phase as described above.

3.1 Algorithm I: Affected-By-Category

Our algorithm AFFECTED-BY-CATEGORY relies solely on road-category information to identify edges that are likely to be affected by delays. We make two assumptions. First, exploiting road-categories leads to a good classification of the nodes into urban and rural regions. Second, most people use faster roads to travel between rural and urban regions. Thus, delays occur mainly on those road-categories, which allow for a fast traveling.

Preprocessing. We locate urban regions within the network by assuming that they are connected by edges e of certain road-category, i.e., $cat(e) \in \zeta_{urban}$, where $\zeta_{urban} \subseteq \zeta$ denotes a set of road categories that is considered to be urban. Additionally, the length of those edges e that are of urban category must not exceed a maximal length $len(e) \leq mDist$. The urban regions are defined by the connected components of the subgraph containing only these edges. They are computed by a modified breadth first search (BFS), which follows only the constrained edges. The urban regions of a road network identified this way are candidates for being towns.

In reality a town is not only connected by slow urban roads. Often faster roads are built to speed up travel times within, out of and into towns. Our simple algorithm omits these roads as otherwise only few quite big towns are found. We reduce the generated fragmentation in the next step. The pseudo-code of the REDUCE-FRAGMENTATION procedure is listed in Algorithm 1. We use a local neighborhood search (BFS) that is limited to a specific number of hops *hLim* to visit nearby nodes and look for their town candidate membership. If the number of neighbors belonging to the same town candidate exceeds a specific ratio *nRatio*, we consider this node to be part of that town candidate too. We store the update information but defer the update itself for a later batch update to prevent towns from growing arbitrarily large.

After the fragmentation has been reduced, we compute the size of each town candidate. If the size exceeds a limit *tThresh*, we consider it to be a town t. Additionally, we store which of the town nodes are boundary nodes by checking whether they are adjacent to nodes outside the town.

Algorithm 1: Reduce-Fragmentation

Input: $G(V,E)$, hop limit *hLim*, neighbor ratio *nRatio*, connected components inform. *CC*
Output: Towns T

```
1 U ⟵ ∅;
2 foreach v ∈ V do
3 |  neighbors ⟵ NeighborhoodSearch(v, hLim);
4 |  maxfrac_i(v) ⟵ largest fraction of neighbors in same CC i;
5 |  if size(maxfrac_i(v)) ≥ nRatio then U ⟵ U ∪ pair(v,i);
6 foreach u ∈ U do CC(u) = i;
```

Urban. In the urban phase we expect people who live in urban regions to travel mainly within that area. This is done all over the day and thus, full-time profiles have to be assigned to edges. To find intensely used roads inside a town, we perform shortest path queries between all boundary nodes B of each town $t \in T$ and assign full-time profiles to the edges of the shortest paths found.

Rural. The regional influence of towns is the basis of the rural phase. Commuters will drive in the morning into and in the evening out of the town. During their travel they have to pass at least one boundary node of towns. We assume that the regional attraction of a town is reduced with the distance people have to travel to reach the town, and we expect travelers to use faster roads to quickly reach their destination. Thus, the idea is to push a certain amount of delay along roads of a specific category starting from the boundary nodes of each town. In each step the delay decays until the process stops.

The set of delay categories used for pushing delays is a subset of all available road categories, i.e., $\zeta_{delay} \subseteq \zeta$. In our generation process we used all road categories that have a faster traveling speed than the urban road categories.

First, the so called *capacity* $c = 10 \cdot size(t)$ of each town is computed, which models the amount of people traveling from and into town. Then, for each town t, the subroutine DAMPENINGBFS[MORNING/AFTERNOON] (t, ζ_{delay}, c, *cDamp*, *rDamp*) is executed. It first sets the interior nodes of the town to be visited to not accidently run into the interior part. Then, the capacity c is equally distributed among the boundary nodes B of t. Depending on the subroutine's type (morning/afternoon), DampeningBFS follows edges (incoming/outgoing) of a given category ζ_{delay} and assignes profile types to the visited edges (MOR/AFT). Every assignment of a profile type to an edge uses up a constant factor $cDamp = 1.0\%$ of the remaining capacity as well as a dynamic part $rDamp = 0.5\% \cdot edgeLength$. The subroutine ends when the capacity is depleted or falls below the threshold $cLim = 100$. In our experiments, we used the constants given above.

The advantage of the algorithm AFFECTED-BY-CATEGORY is that individual capacities and categories can be chosen. Thus, it is possible to model diverse behavior, e.g., short, mid or long distance commuters. A drawback of this approach might be that roads do not have the same category from their start to their end. Thus, the algorithm might not be able to use up a chosen capacity. Additionally, a harsh changeover of the assigned profiles at the boundary nodes of towns can be observed.

3.2 Algorithm II: Affected-By-Region

To overcome the limits of AFFECTED-BY-CATEGORY, we propose another algorithm that involves random shortest path queries between the urban catchment of a town and its interior. The preprocessing and urban phase are the same as in Section 3.1.

Rural. In the rural phase we first identify an area around the town that we expect to be its corresponding urban catchment. The size ℓ of this region is computed using the urban catchment ratio *UCR* by $\ell = UCR \cdot size(t)$. Again we assume that commuters travel from this region into the town. To model this behavior, we perform Dijkstra queries from randomly selected nodes of the *ring* around the urban catchment to randomly selected urban nodes of the town. The ring consists of the neighbors of the nodes after ℓ nodes are found by a local exploration starting at the boundary nodes of the town. A fraction *RQF* of these ring nodes are selected to perform shortest-path queries to randomly selected nodes in the town and back. Profile types are assigned accordingly.

The algorithm AFFECTED-BY-REGION covers much better individual behavior and overcomes the road-category specific limitation of AFFECTED-BY-CATEGORY. However, the algorithm does not model long-distance commuter behavior because it relies on local subroutines. Nevertheless, this can easily be overcome by computing random town-to-town shortest-paths.

3.3 Algorithm III: Affected-By-Level

So far we used road-category information of the source data to locate regions that correspond to towns. The Algorithm AFFECTED-BY-LEVEL uses solely coordinate information of the underlying road network for this purpose.

We assume that towns are well connected and modeled in large detail. Thus, the bounding boxes of urban nodes of a limited local neighborhood search are expected to be small whereas the bounding boxes of rural nodes are rather large. We use the normalized reciprocal value of the occupied area and call it *level* of the node, which is a function $level : V \to [0.0, \dots, 1.0]$. Thus, a high level corresponds to a vertex in an urban region whereas a low value is likely to lie in a rural region.

Preprocessing. In the preprocessing phase we compute the level of each node of the road network to allow for a classification of the nodes into urban or rural ones. The ratio of how many nodes are classified into the urban set can be specified by the parameter *urbR*. First, the local neighborhood of each node $v \in V$ is explored until a limited number *BBL* of *neighbors* is found. Afterwards, the bounding box of the *neighbors* is computed and stored in $area(v)$. Then, the level $lev(v) = 1/area(v)$ of each node is computed. This data is then sorted in ascending order. The value of $urbR \cdot |V|$ determines the index of the entry in the sorted dataset that contains the *urbanThreshold*. Next, for each node $v \in V$, $lev(v) \leq urbanThreshold$ is evaluated. If true, the node is assigned to the rural set R of nodes, otherwise to the urban set U of nodes. Note that in this algorithm we do not model towns.

Urban / Rural. Now we follow the intuition that commuters in most cases travel locally depending on their level. In the morning people travel from rural regions into urban regions and back in the evening. The urban and the main phase are incorporated into the

Algorithm 2: Affected-By-Level

```
  Input: G(V,E), ∀v ∈ V : level(v) ∈ [0,1], R, U, query fraction QF, settled nodes limit SNL
  Output: G(V,E), e ∈ E : p(e) ∈ P
1 toPerform ⟵ QF · |V|; count ⟵ 0;
2 for count ≤ toPerform do
3     u ⟵ randomNode(V); count ⟵ count + 1;
4     S ⟵ LevelDijkstra(u, SNL);
5     if u ∈ R then
6         v ⟵ select random node with high level of S;
7         Dijkstra(u,v,MOR); Dijkstra(v,u,AFT);
8     else
9         v ⟵ select random node with similar level(r) out of S;
10        Dijkstra(u,v,FUL); Dijkstra(v,u,FUL);
```

AFFECTED-BY-LEVEL Algorithm. The pseudo-code is listed in Algorithm 2. First, the fraction QF of local queries in relation to the number of nodes in the graph is determined. Then, a random node u is chosen until the necessary amount of queries to perform is exceeded. Starting from the node u, the subroutine LEVELDIJKSTRA(u, SNL) returns the shortest-path tree S after SNL nodes have been settled. If the node u is of rural type, a random target node v of high level in S is chosen and a morning profile type attached to the edges of the shortest path from u to v. Afterwards, the edges of the shortest path from v to u get an afternoon profile type assigned. In the case that the level of node u is of urban type, a random node v of approximately equal level is chosen. Afterwards, the edges of the shortest paths from u to v as well as from v to u get an full-time profile type assigned.

A clear benefit of the algorithm AFFECTED-BY-LEVEL is that it is all-purpose because it depends solely on coordinate information of the underlying road network. Thus, if road categories are faulty or even absent or in the case of artificially generated data this algorithm can still be applied. A drawback of this approach is that quite a large fraction of the road network has to be locally explored to achieve a representative profile assignment. As a result quite many edges are affected, which is not true for the commercial data. Hence, many edges have to be filtered out to fit with the statistical properties of the real-world dataset.

4 Experiments

In this section we experimentally assess the usefulness of our proposed algorithms. First, we will report on the parameter tuning done to generate meaningful data. The hereby generated datasets are then compared to the commercial dataset by means of statistical properties in a global and local scope. Furthermore, the shortest-path structure of the datasets is analyzed. Our implementation is written in C++ using the STL but no other additional library. The code was compiled with GCC 4.5 and optimization level 3. All experiments were run using OpenSuse 11.3 on one core of an AMD Opteron 6172@2.1 GHz, 512KB L2 cache and 256GB RAM.

Table 2. Parameters used by our algorithms to compute artificial, time-dependent data for the graph of Germany

Algo	mDist	tThresh	hLim	nRatio	UCR	RQF	urbR	BBL	QF	SNL	F
Category	300	500	5	75%	-	-	-	-	-	-	0
Region	300	750	5	75%	3.0	5%	-	-	-	-	0
Level I	-	-	-	-	-	-	45%	200	20%	400	8
Level II	-	-	-	-	-	-	45%	200	40%	400	8

Table 3. Percentage of the profile types assigned to edges of the time-dependent graphs. Additionally their creation time is given in minutes.

Category	PTV	Category	Region	Level I	Level II
notset	93.00%	92.60%	90.73%	92.13%	80.55%
camel	2.73%	2.19%	1.53%	3.60%	9.33%
morning	1.21%	1.22%	3.40%	2.44%	5.30%
afternoon	1.53%	1.22%	3.40%	1.30%	3.30%
full-time	1.50%	2.74%	0.92%	0.50%	1.52%
time (min)	-	55	72	21	26

Input. In the experiments, the road network of Germany that is provided by the company PTV AG was used. The graph consists of about 4.69 million nodes and 11.18 million edges. An additional dataset contains time-dependency information, which refers to a subset of the affected edges of the graph. We further compressed the time-dependent data and substituted each profile by its representative profile, as identified in Section 2.

Parameters. The algorithms can be fine-tuned using many parameters. In Table 2 we derive for each algorithm a parameter set that fits best with the statistical properties of the profile distribution of the real-world dataset, which is shown in Table 3. An exception is the parameter set Level II, which fits best with the statistical properties of the time-dependent Dijkstra experiment, which we show later. Parameters that are not listed here are treated as constants and are specified in the corresponding algorithm in Section 3. When speaking of *generated datasets* we refer to the datasets generated with these parameters. In our discussion we use the following abbreviations for each of the datasets: PTV, Category (CAT), Region (REG), Level I (L1) and Level II (L2).

Global statistical properties. Table 3 shows the distribution of the representative profile sets of the real-world dataset as well as of the generated datasets. CAT fits best the properties of PTV but contains slightly more FUL profiles. REG contains only a third of FUL profiles compared to CAT and three times the amount of MOR and AFT profiles. An explanation is that REG works similar to CAT but randomly selects nodes inside of towns instead of the towns boundary nodes. Thus, by the specification of our postprocessing step many FUL profiles are overridden with profiles of type MOR and AFT. A similar overriding behavior can be observed for L1. Noticeable is the difference of the occurrences of MOR and AFT profiles for L1 and L2, which originates from different shortest paths for nodes (u,v) and (v,u) in combination with the simple filtering rules

Table 4. Amount of profile types assigned to the shortest path edges that are found by Dijkstra during its execution of 10,000 queries. The results have been averaged.

Algorithm	full-time	evening	camel	morning	∑TDE	not-set
PTV	6.14	26.49	28.36	18.79	79.78	188.99
Category	8.16	35.36	7.73	40.18	91.43	177.33
Region	1.09	32.28	31.63	33.60	98.60	170.16
Level I	2.23	4.80	17.66	13.15	37.84	230.93
Level II	5.89	10.07	41.03	31.93	88.92	179.85

Table 5. Algorithmic properties of 23,000 time-dependent Dijkstra queries performed on the commercial and generated datasets. The results have been averaged.

Graph	sNodes	touEdges	tdEdges	errorRate	rel-av	rel-max
PTV	364722	436399	27608.2	-	-	-
Category	364700	436362	32684.0	22.77%	0.39%	5.88%
Region	364705	436364	37852.8	26.07%	0.45%	5.88%
Level I	364721	436400	29787.9	22.62%	0.43%	5.95%
Level II	364725	436407	73641.3	21.88%	0.56%	9.70%

applied. L2 does not aim at fitting best to PTV but we can see that the ratio of profile categories is preserved. Additionally, the running time of each of the generated datasets is given. Note that a large part of the time is consumed by performing point-to-point shortest path queries. Our algorithms can be accelerated using speed-up techniques where possible.

Local statistical properties. The statistical properties given in Table 1 reflect only the global view on the distribution of time-dependent edges in the generated datasets. To compare this information on a local level, we chose 10,000 source-destination pairs at random, performed Dijkstra queries between them and compared the occurrences of profile types on the shortest path found for each of the datasets. In Table 4 the average number of the encountered profile types during each query are shown. For a better overview, the number of edges affected by time dependency are summed in the column *TDE*. Despite minor outliers the relations of the data seem to be of equal size. The outcome of this experiment leads to the recommendation to use REG as it seems to fit best with the local properties of PTV.

Shortest-path behavior. So far we assessed the generated data to have similar structural properties in global as well as local scope. Next, we focus on the algorithmic behavior in order to show that an evaluation of shortest-path algorithms on the generated datasets gives similar results as for the real-world dataset. In Table 5 we present the shortest-path properties of 23,000 Dijkstra queries. In particular, these are the number of settled nodes (sNodes), touched edges (touEdges) and time dependent edges (tdEdges). Additionally, the error rate (errorRate) is given, which specifies the amount of queries that had a different length compared to the referential distance computed on PTV. The average difference between the actual computed distance and the reference distance of the real-world graph is also presented (rel-av). Furthermore, the relative

maximal distance difference is shown (rel-max). The experiment indicates that all of our time-dependency generation algorithms lead to a similar behavior of the shortest path algorithm applied and thus are qualified to generate meaningful time-dependency datasets.

5 Conclusion

We presented the first algorithms to generate realistic time-dependency information for large-scale road networks of continental-size. The scenario we deal with consists of systematic delays on a daily basis within road-networks, which omits unexpected events like accidents, weather or holidays. By the analysis of a commercially used but confidential time-dependent road network of Germany[3], we found a set of representative profiles and deduced a way to classify nodes and edges of the road network in such a way that it is easy to compute edges that are likely to be affected by the aforementioned set of representative profiles. Hence, our algorithms search for urban areas within static road-networks by the utilization of either road-categories or coordinates. Assuming commuters behave in predictable ways depending on their location within the road network, we compute edges that are likely to be affected by time dependency. The experimental study shows the usefulness of the generated dataset in a comparison to the real-world dataset by means of the statistical properties of the generated graphs as well as algorithmic shortest path behavior.

Our work allows experimenters to validate algorithms for time-dependent point-to-point queries on more realistic data than it was previously possible. The running times of our algorithms are practical in the sense that using reasonable parameters, continental-sized graphs can be handled within a few hours. The algorithms incorporate many degrees of freedom that allow for an adaption to specific applications. The running times of the algorithms can be accelerated by replacing Dijkstra's algorithm with speed-up techniques where possible.

Acknowledgments. We thank Reinhard Bauer, Daniel Delling, Ignaz Rutter and the anonymous referee for the valuable discussions and for suggestions how to improve the appearance of this work.

References

1. Flötteröd, G.: Traffic State Estimation with Multi-Agent Simulations. PhD thesis, Technische Universität Berlin (2008)
2. Delling, D.: Time-Dependent SHARC-Routing. Algorithmica (July 2009); Special Issue: European Symposium on Algorithms 2008
3. Nannicini, G., Delling, D., Liberti, L., Schultes, D.: Bidirectional A* Search for Time-Dependent Fast Paths. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 334–346. Springer, Heidelberg (2008)
4. Batz, G.V., Geisberger, R., Neubauer, S., Sanders, P.: Time-Dependent Contraction Hierarchies and Approximation. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 166–177. Springer, Heidelberg (2010)

[3] Provided to us by the company PTV AG.

5. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In: Lerner, J., Wagner, D., Zweig, K.A. (eds.) Algorithmics of Large and Complex Networks. LNCS, vol. 5515, pp. 117–139. Springer, Heidelberg (2009)
6. Delling, D., Wagner, D.: Time-Dependent Route Planning. In: Ahuja, R.K., Möhring, R.H., Zaroliagis, C.D. (eds.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 207–230. Springer, Heidelberg (2009)
7. Kerner, B.S.: The Physics of Traffic. Springer, Heidelberg (2004)
8. Bar-Gera, H.: Traffic assignment by paired alternative segments. Transportation Research Part B: Methodological 44(8-9), 1022–1046 (2010)
9. Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.): The Shortest Path Problem: Ninth DIMACS Implementation Challenge. DIMACS Book, vol. 74. AMS, Providence (2009)
10. Lloyd, S.: Least squares quantization in pcm. IEEE Transactions on Information Theory 28(2), 129–137 (1982)
11. Bauer, R., Krug, M., Meinert, S., Wagner, D.: Synthetic Road Networks. In: Chen, B. (ed.) AAIM 2010. LNCS, vol. 6124, pp. 46–57. Springer, Heidelberg (2010)

An Empirical Evaluation of Extendible Arrays

Stelios Joannou and Rajeev Raman

University of Leicester, Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH

Abstract. We study the performance of several alternatives for implementing *extendible* arrays, which allow random access to elements stored in them, whilst allowing the arrays to be grown and shrunk. The study not only looks at the basic operations of grow/shrink and accessing data, but also the effects of memory fragmentation on performance.

1 Introduction

Dynamic (internal-memory) data structures are ubiquitous in computing, and are often used in on-line, continuously running, software that responds to external events (such as "daemons"). Many classical data structures (heaps, dynamic trees etc), are developed in the *pointer machine* model [17]; this paper is not primarily concerned with these, but with the rapidly increasing number of RAM dynamic data structures (e.g [1]) that have been recently proposed, particularly *succinct data structures* [12,4,11,5,13]. An important feature of these data structures is that they repeatedly allocate and deallocate variable-sized pieces of memory. The memory usage can be measured in two ways:

– In the *memory manager* model the algorithm calls built-in "system" procedures *allocate* and *free*. The procedure *allocate*(x) returns a pointer to the start of a sequence of *contiguous* (unused) memory locations, and increases the memory usage of the algorithm by x units. The procedure *free*(p) takes as an argument a pointer p to a contiguous block of memory locations that was previously *allocated*, and frees the entire block; the memory usage of the algorithm reduces by the requisite number of units.
– In the classical RAM memory model, the algorithm has access to memory words numbered $0, 1, 2 \ldots$. The space usage at any given time is simply $s + 1$ where s is the highest-numbered word currently in use by the algorithm [6].

Although many dynamic succinct data structures [4,9,2] work in the memory manager model, this model does not charge the data structure for space wastage due to *external fragmentation* [16, Ch 9]. It is known that if N is the maximum total size of blocks in use simultaneously, any memory manager needs $\Omega(N \lg N)$ words of memory to serve the requests in the worst case [14,15,10]. (If data is always allocated in fixed-size chunks, there is no serious issue with fragmentation; we also do not consider situations where the memory manager can move an already allocated chunk of memory to a different address.)

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 447–458, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Fragmentation is problematic for a number of reasons. If the memory allocator is directly allocating physical memory, then fragmentation results in significant underuse of physical memory. Of course, most computing devices run operating systems that provide per-process virtual memory, but this is not universal: operating systems such as Android do not support virtual memory, and this appears to be relatively widespread when the secondary storage is based on solid-state technology, due to the current tendency for upgrades to degrade solid-state memory[1]. Even when virtual memory is supported, it is not axiomatic that virtual memory is unlimited — a notable example is the Java VM, which is limited to 2GB of virtual memory. Finally, when virtual memory is effectively unlimited (as it would be on a 64-bit machine), when the data being stored is close to the physical internal memory on a machine, fragmentation may lead to "thrashing", and on smaller input sizes, poor usage of TLB.

Unfortunately, the memory-manager model is the *only* memory allocation method available for normal application programmers, and it is inconvenient (as in [11,5,13]) to simulate the RAM memory model through the memory-manager model (in practice such simulation is impossible if the data structure is to be used as part of a large, complex application). Our aim, therefore, is to find *fragmentation-friendly* dynamic data structures, which (ideally) achieve fragmentation-friendliness through *self-tuning*, and not by means of parameters that the user sets, since these parameters may be highly data-dependent (e.g. they may depend upon the relative amount of textual and markup data in an XML document, the distribution of keys to be hashed in a dictionary etc.).

Collections of extendible arrays. We focus on the above issues in a *collection of extendible arrays* (CEA), which is arguably the simplest dynamic random access data structure, but can be used to build complex data structures [11,13]. A CEA maintains a collection of *extendible arrays* (EAs); each EA in the collection is a sequence of n records. Each record is stored in a single word and is assigned a unique index between 0 and $n-1$. The operations supported are:

- $\mathsf{create}(r)$: create an new empty EA and return its name,
- $\mathsf{destroy}(A)$: destroy the EA A, and
- $\mathsf{access}(i, A)$: access (read/write) the record with index i in the EA A,
- $\mathsf{grow}(A)$: if the EA A currently has n records, add a new record to the end of A with index n.
- $\mathsf{shrink}(A)$: if the EA A currently has n records, delete the record in A with index $n-1$ (the last record).

Although there have been several studies of memory fragmentation in general [3,8,7], we believe this the first study where the effect on fragmentation of a series of allocations/deallocations by a *specific* data structure is studied.

[1] This is especially problematic when the amount of secondary memory is limited, as memory locations will be written to repeatedly by the virtual memory system.

2 Data Structures

We now describe our data structures. A CEA is represented by a vector (as described below) which contains pointers to the individual EAs; the handle of an EA is simply the index in this vector that contains the pointer to the EA. We consider the following implementations of an individual EA:

Vector. This is the standard data structure, which stores an EA with n records in an array of size at most $2^{1+\lceil \log_2 n \rceil}$ records. To handle an intermixed sequence of grow and shrink operations, a rule for resizing the array might be as follows: double the array size whenever there is no more room to accommodate a grow and halve the array size whenever a shrink causes it to become less than 1/4 full.

Remarks. The time for access is worst-case $O(1)$, grow and shrink take $O(1)$ amortized time each and and create/destroy take $O(1)$ time each. However, a vector of size n may have internal fragmentation of $\Theta(n)$ words[2]. Furthermore, assuming a first-fit allocator, it is easy to come up with a sequence of operations that yields n vectors of total size $O(n)$ records that occupy a range of memory addresses spanning $\Theta(n \log n)$ words (details omitted).

Simple. To reduce the internal fragmentation, the simplest idea is to choose a fixed integer parameter $b > 1$ (ideally a power of 2). Records are stored in fixed-size *data blocks* of b words each. For each EA with size n, we store a vector (called the *index block*) that contains $\lceil n/b \rceil$ pointers to each data block; to perform access(i), we access the (i mod b)-th entry in the $\lfloor i/b \rfloor$-th data block.

Remarks. This gives $O(1)$ worst-case time for access and $O(1)$ amortized time for grow and shrink, and $O(1)$ time to create and destroy empty EAs. The use of equal-sized data blocks means that a CEA built upon this EA is less susceptible to external fragmentation. The index block occupies $O(n/b)$ words, this overhead can be minimized by choosing a large value of b. However, if the collection contains a significant proportion of small (size $\ll b$) EAs, there could $\Theta(b)$ words of internal fragmentation per EA, and the internal fragmentation could be even more than for the vector CEA. Thus, the parameter b must be chosen based upon knowledge of the way the DS will be used (which may not be available), and this DS is not "self-tuning". Furthermore, from an asymptotic viewpoint, the fact that index blocks are $\Theta(n)$ in size may mean that external fragmentation caused by index blocks is relevant.

Brodnik. In [4] a vector of size n is divided into consecutive (conceptual) *superblocks* of size $1, 2, 4, \ldots, 2^{\lfloor \log_2 n \rfloor}$. A superblock of size 2^k is represented as up to $2^{\lceil k/2 \rceil}$ data blocks of size $2^{\lfloor k/2 \rfloor}$ each, and memory is only allocated for non-empty data blocks. An index block contains pointers to all data blocks and is represented as a vector. The access(i) function is a little complex:

[2] By *internal fragmentation* we mean memory allocated by a data structure but not used, analogous to the operating systems term [16]).

access(i):
1. Let r denote the binary representation of $i+1$, with all leading zeros removed.
2. The desired element i is element e of data block b of superblock k, where:
   (a) $k = \lfloor \log_2(i+1) \rfloor$,
   (b) b is the $\lfloor k/2 \rfloor$ bits of r immediately after the leading 1-bit, and
   (c) e is the last $\lfloor k/2 \rfloor$ bits of r.
3. Let[3] $p = 2^{\lfloor k/2 \rfloor} + 2^{\lceil k/2 \rceil} - 2$.
4. Return the e-th element of the $(p+b)$-th datablock.

Remarks. Brodnik et al. [4] show how to implement access in $O(1)$ worst-case time. The amortized time for grow and shrink is clearly $O(1)$, and $O(1)$ time is needed to create and destroy empty EAs. It is easy to see that "wasted" space (internal fragmentation plus the index block) is $O(\sqrt{n})$ words. Brodnik et al. [4] show that this level of wasted memory is optimal. However, it is possible to give a sequence of grow and shrink operations that creates $O(n)$ vectors of total size $O(n)$, but occupying $\Theta(n \log\log n)$ words of memory (details omitted).

Modified Brodnik. A modification of Brodnik et al.'s data structure is as follows. All data blocks in a given EA are of the same size b (which is a power of 2), initially $b = 2$. There is initially an index block of size i (also a power of 2), initially $i = 1$. A grow or shrink adds/deletes elements from the last data block, allocating a new data block or freeing a newly-empty one, as needed. Consider now a sequence comprising solely of grow operations. If the index block is full, we alternate between two courses of action: doubling i and doubling b; in the latter case we take pairs of existing data blocks, and copy their data into a newly allocated data block of size $2b$, and free the existing data blocks (this has the effect of making the index block half-full). For a mixture of intermixed grow and shrink operations, if the index block occupancy drops below $1/4$ after a shrink we undo the last "adjustment" operation (i.e. we halve b or i, whichever variable was doubled most recently). The access operation works as in Simple.

Remarks. This gives $O(1)$ worst-case time for access and $O(1)$ amortized time for grow and shrink, and $O(1)$ time to create and destroy empty EAs. However, the CPU cost of the access instruction is significantly lower. Again, the wasted space is $O(\sqrt{n})$ words and, as with Brodnik, it is possible to give a sequence of grow and shrink operations that creates $O(n)$ vectors of total size $O(n)$, but occupying $\Theta(n \log\log n)$ words of memory.

Global Brodnik. Both Brodnik-style data structures above potentially suffer from external fragmentation when used in a CEA. This is because different EAs in the CEA will have different data block sizes (we ignore external fragmentation due to index blocks since the index blocks are typically a small overall component), so a mixture of block sizes will typically be in the process of allocation/deallocation. We now use some ideas from [13] to "self-tune" block sizes. If t is the number of EAs currently created, N is their total size, and b the current

[3] The formula $p = 2^k - 1$ in [4] is (clearly) wrong: there are $O(\sqrt{n})$ data blocks.

block size, then the worst-case internal fragmentation is $O(bt)$, and that due to the index blocks is $O(t + N/b)$. Balancing the two gives the optimal block size as $b = \Theta(\sqrt{N/t})$. The algorithm tries to maintain an ideal block size of $c\sqrt{N/t}$ for some constant $c > 0$, and whenever the real block size is more than a factor of two away from this "ideal" value, it is either doubled or halved, resulting in a re-organization of all EAs in the CEA.

Remark. The time for access is clearly $O(1)$, and in [13] it is shown that the amortized time for grow, shrink and create is $O(1)$; however, this analysis assumes that the number of EAs in existence at any given time is within a constant factor of the maximum number of EAs that were ever in existence in the past. The internal fragmentation is clearly $O(\sqrt{Nt})$ words; representing each EA individually using Brodnik would lead to internal fragmentation of $O(\sum_{i=1}^{t} \sqrt{n_i})$ words, where n_i is the size of the i-th EA, which is better than $O(\sqrt{Nt})$ unless all EAs have roughly the same size.

3 Experimental Evaluation

The aforementioned data structures have been implemented in C++ and a variety of tests were conducted to study the speed as well as the memory usage/fragmentation of the implementations together with the C++ STL vector, which we now describe. The test machine that was used to run these tests was a Intel Core 2 Duo 64-bit machine with 4GB of main memory, 3.16GHz CPU and 6MB L2 cache, running Ubuntu 10.04.1 LTS Linux. The compiler version was g++ 4.4.3 with optimization level 3. The CEAs all stored 4-byte integer records; note that pointers are 8 bytes each. To measure the memory, both virtual memory (VM) and resident memory (RES) that was used by the tests, `/proc/file/stat` was used. For the speed tests `clock()` method was used to measure CPU time and the `/usr/bin/time` command for wall time.

3.1 Implementation Details

For all of these DS, except Global Brodnik, a common collection manager class was used, to allow multiple instantiations of EAs, choosing the DS using compiler options. The collection manager uses an array of pointers to store the memory locations of each instance of the DS. Every time the array is full it doubles its size. The EAs are allocated in memory using the `new` keyword.

Vector. We used the standard STL implementation, which uses doubling if the underlying array is full, but when elements are removed, the underlying array size does not change in any way.

Brodnik. This implementation of this DS is based on the original paper. To optimize the speed of access (and also shrink and grow), a number of values are stored in the header block of this DS, giving a relatively large header size of 41 bytes. Further, to optimize access(i), some operations (e.g. $\lfloor x/2 \rfloor$, $\lceil x/2 \rceil$) were

written using bitwise operations. To compute $\lfloor \log_2 x \rfloor$ (the left-most set bit in an integer x) the folklore trick of casting x to a float is used. We access this memory as an integer, use bitwise operations to extract the exponent, then subtract the bias, and the result is the position of the left-most set bit. Finally, a table of size at most 64 integers was used to map the number of the superblock that the i-th record is located in, to the number of data blocks prior to the specified superblock. These optimizations greatly increased the speed of access(i).

Simple. This DS is implemented with the data block size (which must be a power of 2) being a constructor parameter. In the access function, operations such as division by b and modulo b are implemented by shifts and masks, respectively. We used $b = 2^6 = 64$ throughout in our tests. Again a number of header variables are used and the header block size is 29 bytes.

Modified Brodnik. The access(i) operation is similar to Simple, it uses masking and shifting to get the location of element in a data block and the location of that data block in a super block. Since the size of these data blocks changes over time as elements are added or removed, a static array of masks was used. Since growing the index block and data block alternates, a boolean was used to check what was doubled last (the index block size or the data block size). This DS has a header size of 30 bytes.

When the data blocks need to double, every two data blocks are merged into a new one with double the size. This new data block is stored in the already existing index block starting from the beginning (thus avoiding the creation of a new index block). Similarly where access is worst-case $O(1)$, grow and shrink would take $O(1)$ amortized time each and and create/destroy applied to a new EA/empty EA would take $O(1)$ time each. However, a vector of size n may have internal fragmentation of $\Theta(n)$ words. Furthermore, assuming a first-fit allocator, it is easy to come up with a sequence of creates, grows and shrinks that yields n vectors of total size $O(n)$ records, occupying a range of memory addresses spanning $\Theta(n \log n)$ words in total. When shrinking, either the size of the index block or the data block will be halved. When data blocks need to shrink, one data block is split into two and this is done by storing the new bigger data block into a new index block of the same size as the original one. The old data blocks and index blocks are subsequently deleted.

Global Brodnik. Each individual EA has a header size of 25 bytes. The collection maintains the total number of elements t in all the instances of the EAs that it contains. We derive from the current data block size b (a power of 2) an upper bound $U = 2b$ and lower bound $L = b/2$. After each grow/shrink we use calculate an ideal block size $\hat{b} = \sqrt{t/N}$, where N is the number of EAs. We maintain the condition that $L < \hat{b} < U$: if this condition is violated then the data block size is doubled/halved, along with U and L, to restore this condition. We avoid doing a square root calculation every time there is a grow or shrink by checking if $t/N \geq U^2$ for the upper bound and similarly for shrink. The values U^2, L^2 are recomputed every time U and L change.

The access method is the same as modified Brodnik and an array of masks is used. The index block of an individual EA is doubled when it gets full, either to accommodate one new data block, or because the size of the data blocks is halved. An index block is halved when its occupancy drops below a quarter of its capacity (by a shrink on an individual EA).

3.2 Speed Tests

We tested the time for access(i) (specifically a read — writes were not tested). In all cases EAs were created *sequentially*, i.e., the i-th EA was created and grown to its final size before creating and growing the $(i+1)$-st EA. We considered two access patterns: *sequential* and *random*. For the sequential access test elements were accessed in the order in which they were grown. In the random access test, we instead made uniform random accesses equal to the number of elements in the CEA. The random test was essentially run only in the case where all EAs in the CEA are equally sized, and the number of EAs and elements per EA are both powers of two. In this case we used one call to the `lrand48()` method in the C++ `cstdlib`. This generates a number in the range of $[0, 2^{31})$: we use the most-significant bits to select an EA and the least-significant bits to select an element within that EA. This avoids making two calls to `lrand48()` (which is relatively slow), but limits the total number of elements t that can be used for this test to 2^{31}. This was not a limitation as data sizes such as these would have exceeded the RAM of our machine.

A variety of values of N (the number of EAs) and k (the size of each EA) was used. The values used were $N = 16$ and $k = 16777216$ (a few large EAs), $N = k = 16384$ and $N = 2097152, k = 128$ (many small EAs, relevant to some

Table 1. Growing time, Sequential and Random access time test results (in seconds)

EAs x Elements	DS	Grow	Sequential	Random
16 x 16777216	Vector	2.38	0.25	22.65
	Brodnik	2.93	1.90	28.66
	Simple	1.87	0.31	40.53
	Modified Brodnik	1.69	0.29	20.63
	Global Brodnik	4.95	0.33	23.96
16384 x 16384	Vector	1.90	0.25	24.03
	Brodnik	3.12	1.87	57.46
	Simple	1.85	0.32	44.35
	Modified Brodnik	2.39	0.30	48.05
	Global Brodnik	4.93	0.34	44.46
2097152 x 128	Vector	3.12	0.29	44.69
	Brodnik	6.31	2.09	86.45
	Simple	2.11	0.43	56.28
	Modified Brodnik	6.21	0.58	54.04
	Global Brodnik	6.26	0.48	58.26

succinct dynamic data structures). Each test was run five times and the average of these times is included in this paper. Table 1 gives the results.

As can be seen, all the data structures are significantly faster than Brodnik for sequential access. This is very much as expected (and Brodnik is not particularly "slow" in absolute terms). Also vector is slightly faster than other EAS in all the tests. The random tests show more interesting structure. In the first test, most data structures are similar except for Simple, which is a bit slower. All the data structures used for these tests except for the vector require two memory accesses to retrieve the required element due to indirection cause by the index blocks. This is the main reason why in general the vector is faster than the other DS that were tested. However, in the first test the size of the index blocks in all but Simple are very small (they grow as $\sqrt{n}$, where n is the size of an individual EA) and so fit in cache. However, this pattern is not repeated in the other tests, since the overall size of the index blocks (as a proportion of data blocks) increases as n decreases. There is a slight advantage to the Global and Modified Brodnik in terms of access times, we believe that this may be because the regular rearrangement of data in Global and Modified results in more compact storage; but this requires further investigation.

3.3 Memory Usage Results

Worst Case for Brodnik DS. We now discuss a potential worst-case scenario for the Brodnik DS. The scenario is constructed assuming that there is some "first-fit-like" behavior in the memory manager and will be tested experimentally against the real-life Linux allocator.

The Brodnik DS, as mentioned before, has a header block, an index block and data blocks grouped into virtual superblocks. The test proceeds in rounds $0, 1, 2 \ldots$. In round i, N_i EAs of size k_i are created sequentially (see beginning of 3.2); in rounds $i > 0$ this creation is accompanied by shrinking (in a round-robin manner) the EAs created in round $i-1$. We choose j_0 to be an even integer and let $k_0 = 2^{j_0+1} - 1$; in subsequent iterations we take $j_{i+1} = 2j_{i+1} + 4$ and $k_{i+1} = 2^{j_{i+1}+1} - 1$. We always maintain $N_0 k_0 = N_1 k_1 = N_2 k_2$ and so on, so that the total number of elements in the CEA stays the same.

The reason why this pattern may result in fragmentation is as follows. We hope that if we sequentially allocate N_i EA of size k_i, the space between the header blocks of EAs will be approximately equal to k_i. Note that for EAs of size $2^{x+1} - 1$, the last superblock is of size 2^x, and the size of the data blocks in the last superblock is $2^{\lceil \frac{x}{2} \rceil}$. Thus, in the next round, the data blocks in the last two superblocks of the newly created EAs will be of size $2^{j_i+2} > 2k_i$. Thus, we hope that all these data blocks (which total 3/4 of the EAs created in the $i+1$-st phase) will be allocated in "fresh" memory.

In the test, we chose $j_0 = 4$, giving $k_0 = 31, k_1 = 8191$ and $k_2 = 2^{29} - 1 = 536870911$. Assuming that $N_2 = 1$, this would imply that $N_0 = k_2/k_0$, but our machine was unable to allocate so many EAs. Hence we chose $N_0 = 2^{22}$, $N_1 = N_0 k_0 / k_1$, $N_2 = 1$, and $k_2 = N_0 k_0 \approx 2^{27}$. The results are shown in Table 2.

Table 2. The results of the Brodnik DS worst case

i	N_i	k_i	VM (GB)	RES (GB)
0	4194304	31	2.46	2.45
1	15873	8191	3.03	2.99
2	1	130023424	3.53	3.08

Although the real size occupied with the data blocks should be close to 507MB, due to the headers of the data structure it reaches the 2.46GB initially as shown above. In subsequent phases, there is an increase of almost 570MB, showing that in each case most of the memory allocated for the data blocks is coming from "new" memory, not memory previously freed, even though the very last EA is not quite as large as needed by the formula.

Random. For this test we start with N EAs sequentially created, each of size k. Then we go through the CEA and `shrink` each EA once. We call this a pass. After each shrink we `grow` one EA. The EA to be grown is selected based on the following rule: the first 20% of the EAs should contain 80% of the elements. This rule is applied recursively so 20% of the first 20% of EAs should contain 80% of the total number of elements of the first 20% of the EAs. We go through all the EAs k times, so the EAs at the beginning of the CEA should be larger and the EAs which have not been grown will have 0 elements.

To run this test the values $N = 2^{16}$ and $k = 1024$ were used. The gradual increase after every pass is shown in Figure 1. Table 3 shows the initial memory usage after creation and growing of the N EAs of size k and the resulting memory usage after this test was run. The important thing to notice in this test is that there has been a significant memory increase in all of the data structures without adding new elements, just by redistributing the elements within the EAs.

Thrashing. For the thrashing test we created $N = 2^{19}$ EAs sequentially, each of size $k = 1200$, equating to about 2.4GB of useful data (recall that our machine has 4GB RAM). We performed random access tests immediately after creation and *after* growing and shrinking arrays as in the the 80-20 test described in

Table 3. Memory usage before and after 80-20 test

DS	Initial		Ending	
	VM (KB)	RES (KB)	VM (KB)	RES (KB)
Vector	277648	266980	692040	603772
Brodnik	388228	377576	628456	523156
Simple	304276	293624	357080	343476
Modified Brodnik	328828	318264	577224	485612
Global Brodnik	328768	318208	372900	357440

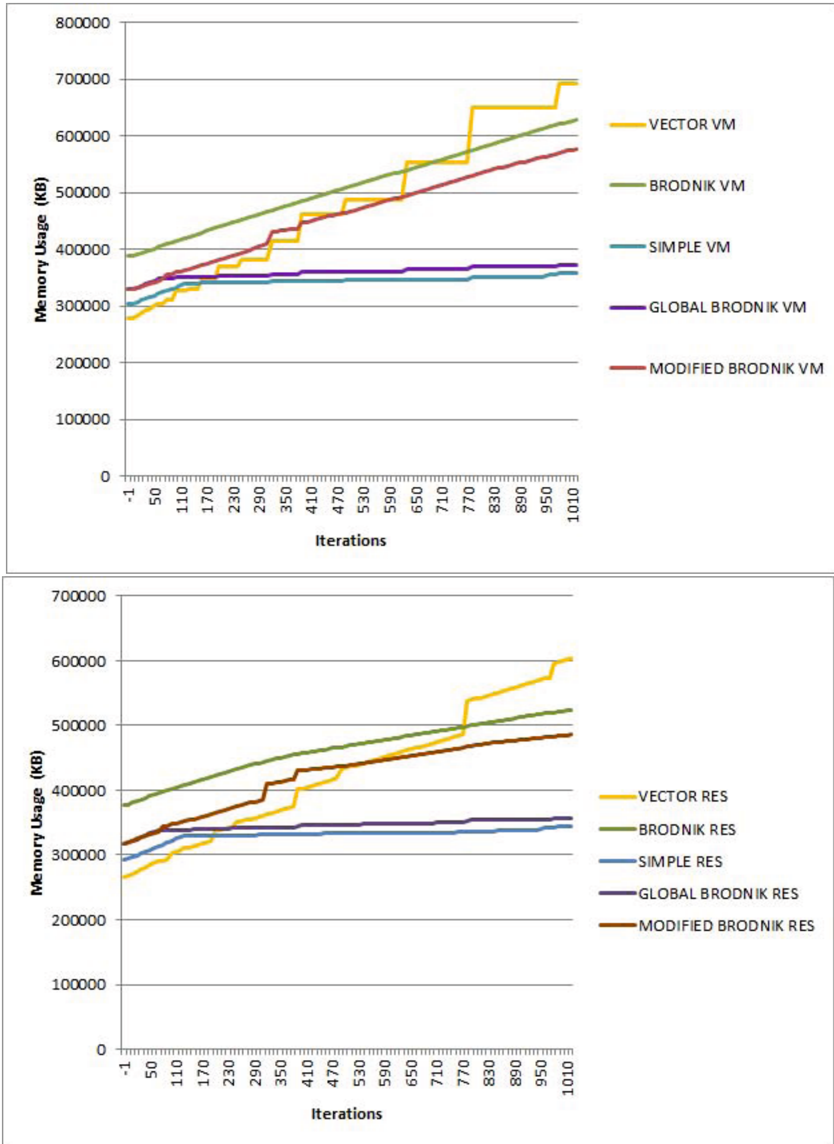


Fig. 1. Virtual Memory (Top) and Resident Memory (Bottom) results of the 80-20 test

Table 4. Memory usage (GB) before and after 80-20 EA modification in thrashing test; CPU and elapsed time for second random access test (s)

DS	Initial		Final		CPU	Elapsed
	VM	RES	VM	RES		
Vector	4.23	3.73	7.34	3.74	40.12	780
Brodnik	3.66	3.65	6.06	3.73	40.19	872
Simple	2.83	2.82	3.20	3.17	28.2	150
Modified Brodnik	3.15	3.14	5.71	3.67	43.28	1988.4
Global Brodnik	3.15	3.14	3.51	3.47	25.52	134.4

Section 3.3[4]. We measured the following: (a) CPU time for the first random access test (b) VM/RES before and after the 80-20 test (c) CPU/elapsed time for the second random access test. The results for (b) and (c) are shown in in Table 4 – we do not report (a) because they are in line with Table 1, except that Brodnik was slower by a factor of 2 than expected (the initial VM was close to the physical memory of the test machine). For (c), Brodnik and Modified Brodnik fell foul of thrashing and took over 14 minutes to complete(thrashing was verified by inspecting CPU usage and page faults using `top`). Vector, despite a high VM, completed, albeit slowly, because it allocates contiguous chunks of memory. Simple and Global Brodnik performed the best in this case.

4 Conclusions

In this paper we have investigated a simple random-access dynamic data structure, the collection of extendible arrays. The standard solution would be to use a number of vectors, but this solution runs into memory fragmentation problems. We have demonstrated a sharp rise in virtual memory usage for the standard solution. We have also conducted some tests that demonstrate that for appropriate data sets that require memory close to the physical memory of the machine, after running the 80-20 test described in section 3.3 the memory requirements were greater than the physical memory of the machine, thus thrashing occured. Unfortunately, the same is true for the data structure proposed by Brodnik et al., which is aimed at solving this problem. We observe that the simple solution of using indirection, together with the so-called "Global Brodnik" seem to avoid this problem, but the simple solution requires parameter setting (which in turn requires knowledge of how the data structure is used) which would appear to preclude it as a general-purpose solution. However, "Global Brodnik" is relatively slow when supporting the `grow` and `shrink` operations, which should be investigated further. Another important task would be to compare their performance on real-life inputs.

[4] With a minor modification: we never let a `shrink` reduce the size of an EA below 200.

References

1. Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. J. ACM 54(3), 13 (2007)
2. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: FOCS, pp. 787–796. IEEE Computer Society, Los Alamitos (2010)
3. Brodal, G.S., Demaine, E.D., Munro, J.I.: Fast allocation and deallocation with an improved buddy system. Acta Inf. 41(4-5), 273–291 (2005)
4. Brodnik, A., Carlsson, S., Demaine, E.D., Munro, J.I., Sedgewick, R.: Resizable arrays in optimal time and space. In: Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 37–48. Springer, Heidelberg (1999)
5. Farzan, A., Munro, J.I.: Dynamic succinct ordered trees. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5555, pp. 439–450. Springer, Heidelberg (2009)
6. Hagerup, T., Raman, R.: An efficient quasidictionary. In: Penttonen, M., Schmidt, E.M. (eds.) SWAT 2002. LNCS, vol. 2368, pp. 1–18. Springer, Heidelberg (2002)
7. Purdom Jr., P.W., Stigler, S.M.: Statistical properties of the buddy system. J. ACM 17(4), 683–697 (1970)
8. Knuth, D.E.: The Art of Computer Programming. Fundamental Algorithms, vol. I. Addison-Wesley, Reading (1968)
9. Lee, S., Park, K.: Dynamic rank/select structures with applications to run-length encoded texts. Theor. Comput. Sci. 410(43), 4402–4413 (2009)
10. Luby, M., Naor, J., Orda, A.: Tight bounds for dynamic storage allocation. In: SODA, pp. 724–732 (1994)
11. Munro, J.I., Raman, V., Storm, A.J.: Representing dynamic binary trees succinctly. In: SODA, pp. 529–536 (2001)
12. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 2001. LNCS, vol. 2125, pp. 426–437. Springer, Heidelberg (2001)
13. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 357–368. Springer, Heidelberg (2003)
14. Robson, J.M.: An estimate of the store size necessary for dynamic storage allocation. J. ACM 18(2), 416–423 (1971)
15. Robson, J.M.: Bounds for some functions concerning dynamic storage allocation. J. ACM 21(3), 491–499 (1974)
16. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 7th edn. John Wiley & Sons, Inc., Chichester (2004)
17. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM, Philadelphia (1987)

Author Index

Abraham, Ittai 230
Althaus, Ernst 340

Berger, André 43
Brisaboa, Nieves R. 136

Cánovas, Rodrigo 136
Claude, Francisco 136

D'Angelo, Gianlorenzo 88
Daniels, Karen 54
Daryabari, Mojtaba 124
de Assis, Igor R. 304
Delling, Daniel 230, 376
de Souza, Cid C. 304

Eppstein, David 364

Festa, Paola 410, 421
Fontes, Dalila B.M.M. 327
Fontes, Fernando A.C.C. 327
Frigioni, Daniele 88
Frinhani, Rafael M.D. 410

Galbiati, Giulia 112
Geisberger, Robert 100
Goerigk, Marc 181
Goldberg, Andrew V. 230, 376
Gonçalves, José F. 421
Gualandi, Stefano 112

Haapasalo, Tuukka 76
Hein, Alexander 218
Heinz, Stefan 400

Joannou, Stelios 447

Katz, Bastian 292
Kaufmann, Michael 267
Kliemann, Lasse 254
Kontogiannis, Spyros 1
Koster, Arie M.C.A. 218
Kottler, Stephan 267
Koukouvinos, Christos 148
Kouri, Tina 157
Kumin, Hillel 33

Landa-Silva, Dario 280
Lavor, Carlile 206
Li, Weiqi 65
Liberti, Leo 206
Longo, Humberto 315
Luxen, Dennis 242

Maculan, Nelson 206
Maffioli, Francesco 112
Malliavin, Therese 206
Martinelli, Rafael 315
Martínez-Prieto, Miguel A. 136
Mateus, Geraldo R. 410, 421
Mehta, Dinesh 157
Meinert, Sascha 434
Minaei-Bidgoli, Behrouz 124
Mucherino, Antonio 206

Nagarajan, Chandrashekhar 169
Naujoks, Rouven 340
Navarro, Gonzalo 136, 193
Nilges, Michael 206
Nourazari, Sara 33

Pajor, Thomas 376
Parsa, Saeed 352
Parvin, Hamid 124
Pecin, Diego 315
Poggi, Marcus 315
Puglisi, Simon J. 193

Raman, Rajeev 447
Resende, Mauricio G.C. 410, 421
Röglin, Heiko 43
Roque, Luís A.C. 327
Rutter, Ignaz 292

Sanders, Peter 242
Schieferdecker, Dennis 388
Schöbel, Anita 181
Schulz, Jens 400
Silva, Diego M. 421
Silva, Ricardo M.A. 410, 421
Silvasti, Panu 76
Simos, Dimitris E. 148
Sippu, Seppo 76

Soisalon-Soininen, Eljas 76
Spirakis, Paul 1
Strash, Darren 364
Strasser, Ben 292

Thaden, Eike 340
Tokgöz, Emre 33

Ülker, Özgür 280

Vahidi-Asl, Mojtaba 352
Valenzuela, Daniel 193
van der Zwaan, Ruben 43
Vetter, Christian 100
Vitale, Camillo 88
Völker, Markus 388

Wagner, Dorothea 292, 388, 434
Werneck, Renato F. 230, 376
Williamson, David P. 169

Yang, Xin-She 21
Ye, Shu 54

Zareie, Farzaneh 352